Project N°: **FP7-231620**

**Project Acronym:** HATS

**Project Title:** Highly Adaptable and Trustworthy Software using Formal Models

**Instrument:** Integrated Project

**Scheme:** Information & Communication Technologies

Future and Emerging Technologies

---

**Deliverable D3.5**

Autonomous Evolving Systems

Due date of deliverable: (T0+48)

Actual submission date: 1 March 2013

---

Start date of the project: **1st March 2009**  
Duration: **48 months**

Organisation name of lead contractor for this deliverable: **KTH**

---

### Integrated Project supported by the 7th Framework Programme of the EC

<table>
<thead>
<tr>
<th>Dissemination level</th>
<th>PU</th>
<th>Public</th>
<th>PP</th>
<th>Restricted to other programme participants (including Commission Services)</th>
<th>RE</th>
<th>Restricted to a group specified by the consortium (including Commission Services)</th>
<th>CO</th>
<th>Confidential, only for members of the consortium (including Commission Services)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PU</td>
<td>Public</td>
<td>PP</td>
<td>Restricted to other programme participants (including Commission Services)</td>
<td>RE</td>
<td>Restricted to a group specified by the consortium (including Commission Services)</td>
<td>CO</td>
<td>Confidential, only for members of the consortium (including Commission Services)</td>
</tr>
</tbody>
</table>

Final version
Executive Summary:
Autonomous Evolving Systems

This document summarizes deliverable D3.5 of project FP7-231620 (HATS), an Integrated Project supported by the 7th Framework Program of the EC within the FET (Future and Emerging Technologies) scheme. Full information on this project, including the contents of this deliverable, is available online at http://www.hats-project.eu.

This deliverable reports on several lines of work that explore the use of the ABS language and framework to support software evolvability, software adaptation, and auto-configuration at runtime, with the goal of producing software systems that evolve in a goal-driven and, as far as possible, autonomous manner.

Two lines of work are followed, on runtime adaptability in networked models with explicit routing, focusing on performance aspects, and on runtime adaptability based on product line engineering, focusing on software configuration aspects.

The first line of work explores the foundations of efficient runtime adaptability by proposing ABS-NET, a new model for object mobility where programs execute on top of a network model where routing is explicitly integrated into the abstract language runtime. This “network semantics” uses a concept of location independent routing to obtain an execution model where a lot of the overhead due to object and task mobility in existing approaches is eliminated. The deliverable presents a network semantics for several sublanguages of core ABS and proves their correctness by establishing soundness and full abstraction in relation to a standard semantics based on rewrite logic.

A distributed simulation framework for ABS-NET has been developed, allowing to simulate network and program configurations of reasonable size (> 100 objects on < 100 network nodes). The simulation framework is exploited to experimentally study the performance of ABS-NET using different load balancing and performance tuning regimes on a selection of basic network and application models, using different network topologies, and a selection of program examples that perform distributed computation in different types of runtime configuration such as stars, rings, trees, and distributed hash tables (DHTs).

The second line of work concerns dynamic adaptation of ABS-based software product lines (SPL’s). Earlier work in HATS has shown how to statically reconfigure SPL models by taking an ABS core model and a set of delta modules and flattening them to obtain an executable core ABS model of the end product. We add support for runtime product reconfiguration to ABS by adding a dynamic representation of product specifications and delta modules and by deferring the flattening process to the runtime. Product reconfiguration takes the runtime representation of a product and applies a set of dynamic deltas to obtain a different product of the SPL. To support this the ABS language has been extended with constructs for modeling software variability using SPL engineering principles focusing on dynamic aspects. Furthermore an illustrative example is presented.

List of Authors

Mads Dam (KTH)
Andreas Lundblad (KTH)
Radu Muschevici (KUL)
Karl Palmskog (KTH)
Thiago Henrique Burgos de Oliveira (FRG)
Contents

1 Introduction ........................................... 5
   1.1 Performance Adaptation ............................. 5
   1.2 ABS-NET ........................................ 6
      1.2.1 microABS-NET and milliABS-NET ............. 6
      1.2.2 Experiments ................................ 7
   1.3 Dynamic Software Product Lines .................. 7
   1.4 Deviations from the DoW ......................... 7
   1.5 List of Papers Comprising Deliverable D3.5 .... 8

2 Location Independent Routing for Process Networks 9
   2.1 Introduction ....................................... 9
   2.2 microABS ........................................ 11
   2.3 Network Semantics: Runtime Configurations ...... 15
   2.4 Correctness ....................................... 20
   2.5 Discussion ....................................... 23

3 Efficient Routing of Futures in Object Networks 25
   3.1 Introduction ....................................... 25
   3.2 mABS .............................................. 26
   3.3 Reduction Semantics ................................ 27
   3.4 Network Semantics ................................ 28
      3.4.1 milliABS-NET Runtime Syntax ................ 29
      3.4.2 Reduction Semantics .......................... 30
   3.5 Normal Forms ...................................... 31
      3.5.1 Normalization ................................ 32
   3.6 Correctness ....................................... 33
   3.7 Scheduling ........................................ 36

4 ABS-NET: Fully Decentralized Runtime Adaptation for Distributed Objects 37
   4.1 Introduction ....................................... 37
   4.2 Core ABS Adaptation ................................ 37
   4.3 Network Model and Semantics ...................... 38
      4.3.1 Operational Semantics ....................... 39
      4.3.2 Assumptions for Adaptability ................ 40
      4.3.3 Node Controller Behaviour ................... 40
      4.3.4 Object Behaviour ............................. 40
   4.4 Adaptation ....................................... 40
      4.4.1 Node Load Balancing ......................... 41
      4.4.2 Minimizing Communication and Other Objectives 41
   4.5 Evaluation ....................................... 42
      4.5.1 Simulator .................................... 42
4.5.2 Scenarios .................................................. 43
4.5.3 Scenario Objectives ..................................... 45
4.5.4 Results .................................................... 45
4.6 Conclusions and Future Work ............................ 47

5 Dynamic Software Product Lines .......................... 50
5.1 Introduction ................................................. 50
5.2 Related Work ................................................ 51
  5.2.1 Dynamic Software Update ............................ 51
  5.2.2 Dynamic Software Product Lines .................. 51
  5.2.3 Related Work within HATS ......................... 52
5.3 Static Variability in ABS .................................. 53
  5.3.1 Feature model .......................................... 53
  5.3.2 Product selection ...................................... 54
  5.3.3 Core module ........................................... 54
  5.3.4 Delta module ........................................... 54
  5.3.5 Configuration .......................................... 55
5.4 Modeling DSPLs ............................................. 55
  5.4.1 Product selection ...................................... 56
  5.4.2 DSPL Configuration .................................... 57
  5.4.3 State Transfer .......................................... 59
5.5 Dynamic Reconfiguration .................................. 59
  5.5.1 Building Self-Adapting Systems .................... 60
  5.5.2 Self vs. Externally Triggered Adaptation ........... 61
  5.5.3 Adaptive Runtime Environment ...................... 61
5.6 Conclusion .................................................. 61

6 Conclusion .................................................... 62

Bibliography ..................................................... 63

Glossary .......................................................... 70

A Adaptibility-Simulated Core ABS Programs ............... 72
  A.1 IndependentTasks.abs .................................... 72
  A.2 Ring.abs .................................................. 72
  A.3 Star.abs .................................................. 74

B Paper 1: Location Independent Routing in Process Network Overlays 76
C Paper 2: Efficient and Fully Abstract Routing of Futures in Object Network Overlays 118
D Paper 3: ABS-NET: Fully Decentralized Runtime Adaptation for Distributed Objects 170
Chapter 1

Introduction

Software systems are exposed to change for many different reasons:

- Changes in operating conditions. This sort of change can occur, for instance, due to changing performance requirements, changes in application load, or changing operating configurations, for instance due to mobility, changing power requirements, failures, or attacks.

- Changes in execution platform, for instance because hardware or APIs with better performance or new functionality becoming available.

- Changes in the code base, for instance because of patching, code refactoring, or addition or removal of functionality.

To support these different types of changes different approaches are needed. In this deliverable we report on several lines of work that explore the use of the ABS language and the HATS development framework to support software evolvability, software adaptation, and auto-configuration at runtime, with the goal of producing software systems that evolve in a goal-driven and, as far as possible, autonomous manner.

1.1 Performance Adaptation

The first line of work focuses on performance adaptability, specifically the problem of allowing a program to automatically adapt to changes in the configuration of an underlying networked execution platform. This problem has at least the following three dimensions:

- Adaptation mechanisms: First, a set of mechanisms is needed that allows the system to reconfigure in a safe manner. This includes mechanisms such as reconfiguring local execution settings such as priorities and scheduling options, migrating objects and tasks between cores/execution units, and changing memory allocation or channel bandwidth.

- Monitoring, measurement, and instrumentation: Second, a set of mechanisms is needed for controllers and schedulers to determine the system state. This includes routing information, processor and link loads, and other, more global, information such as end to end latencies.

- Adaptation algorithms: Finally, control algorithms are needed that use the adaptation and monitoring artefacts to control execution to meet given systems objectives, here typically performance and safety requirements. Generally the goal is that the system state stays within a desired safe operation perimeter, and such that operation is efficient, i.e., it does not consume more resources than necessary while meeting required performance properties, for instance related to resource utilization or application performance.
Figure 1.1: Control systems view of adaptability

Figure 1.1 fits these components into a familiar control systems setting with the important proviso that “objectives” are (derived from) global systems objectives whereas control decisions are made locally at each processing node, based on local monitored behaviour. This control systems view of adaptability is quite well accepted. Much work in the cloud computing domain, for instance, aims at achieving datacenter scale predictability of latency and bandwidth consumption using dynamic, but centralized control schemes, for problem domains such as VM scheduling, network access, and live migration cf. [45, 44, 55, 2]. The main differentiator with respect to the work reported here is that we take an abstract, semantics-based approach aimed at producing solutions that are fully decentralized, dynamic, and scalable (though we do not claim to have achieved these goals yet).

1.2 ABS-NET

As our main contribution in this part of the deliverable we propose ABS-NET: a new model for object mobility where programs execute on top of a network model where routing is explicitly integrated into the abstract language runtime. This “network semantics” uses a concept of location independent routing to obtain an execution model where a lot of the overhead due to object and task mobility in existing approaches is eliminated. This is important, since existing approaches to object mobility suffer from inefficiencies due to rerouting and message forwarding that typically make them unusable in application scenarios requiring high performance. Our approach could be used to optimize live migration overhead in practical cloud computing settings as well: To the best of our knowledge this type of approach has not yet been attempted.

1.2.1 microABS-NET and milliABS-NET

The network semantics is developed in two steps. In Chapter 2 of the deliverable, we first introduce a small fragment of the core ABS language, microABS-NET, essentially corresponding to a language of asynchronous message passing processes. The microABS-NET language corresponds roughly to the nomadic PICT language by Sewell and his colleagues [78]. The fragment is then enriched in Chapter 3 to capture what is essentially the asynchronous fragment of core ABS, called milliABS-NET. For both these languages, operational semantics are given at an abstract level using a standard rewriting logic approach, in order to determine the reference run-time behaviour of programs when most aspects of physical distribution, including naming, routing, and locality are ignored, similar to the style of semantics considered in other of the HATS project deliverables. For each of the languages, a network semantics is then proposed that localizes program execution to explicit network nodes, connected pairwise by asynchronous, buffered, point to point links. We then analyze the correctness of the network semantics by proving the equivalence of the two semantics in the absence of a scheduler, i.e., where all runtime choices concerning routing updates, object migration, message delivery, and task scheduling are resolved in a fully nondeterministic fashion. Part of this analysis involves proving convergence and self-stabilization type properties. The correctness results are very strong, in one go establishing
a lot of desirable properties such as absence of deadlocks, eventual delivery of messages, as well as critical consistency properties. Moreover, as we show, the results can be used to easily derive simulation results in the presence of scheduling as well.

### 1.2.2 Experiments

In Chapter 4 we report on an extension of ABS-NET to the full core ABS language, as described in [56]. We present a network semantics generalizing the microABS-NET and milliABS-NET semantics, but organized a little differently by splitting the runtime state into two components as follows:

1. A language-independent node controller, responsible for monitoring, exchanging state information with neighbours, and adaptation, making scheduling decisions chiefly by determining when and where to migrate objects.

2. A language dependent interpreter layer, responsible for program execution, delegating all functionality concerning control, communication and synchronization, object migration, and scheduling to node controllers.

A distributed simulation framework for ABS-NET has been developed, allowing to simulate network and program configurations of reasonable size (>100 objects on <100 network nodes). The simulation framework is exploited to experimentally study the performance of ABS-NET using different load balancing and performance tuning regimes on a selection of basic network and application models, using different network topologies, and a selection of program examples that perform distributed computation in different types of runtime configuration such as stars, rings, trees, and distributed hash tables (DHTs).

### 1.3 Dynamic Software Product Lines

In Chapter 5, we present our approach to handle dynamic reconfiguration (runtime adaptation) of ABS code. Dynamic software product reconfiguration is understood as the ability to reconfigure products at runtime, that is, the transformation of a product into another valid product defined by a given software product line (SPL), all without the need to re-compile and redeploy the system. To support this kind of dynamic adaptation, ABS models need to accommodate dynamic changes in their structure and behaviour. Adding this facility complements the static SPL modeling capability of ABS. Static product generation introduced support for configuring a particular SPL product at compile time by taking an ABS core model and a set of delta modules and flattening them to obtain an executable core ABS model of the end product. We add support for runtime product reconfiguration to ABS by adding a dynamic representation of product specifications and delta modules and by deferring the flattening process to the runtime. Product reconfiguration takes the runtime representation of a product and applies a set of dynamic deltas to obtain a different product of the SPL.

The ABS language has been extended with constructs for modeling software variability using SPL engineering principles focusing on dynamic aspects. Furthermore an illustrative example is presented.

If from a technical perspective everything can be changed/adapted at runtime, from the product line perspective only a limited range of runtime changes may make sense and maintain the consistency of a product. In order to avoid breaking product consistency when adapted at runtime, there must be a way to constrain the adaptations that could be performed at runtime. Thus, the possible adaptations performed at runtime must be planned such that the resulting product (after the adaptations are performed) is a consistent product adhering to all feature model constraints and definitions.

### 1.4 Deviations from the DoW

Task 3.5 as formulated in the DoW focuses on three problem domains, namely functionality, performance, and security. To keep the scope of Task 3.5 manageable it was decided early on to delegate security concerns
regarding adaptability and self-monitoring to the relevant Tasks 4.1 and 3.4. The task has also had a more explorative character than envisaged in the DoW, since the development and analysis of the ABS-NET model turned out to be a larger enterprise than foreseen. For this reason, the ABS-NET simulator got operational too late in the task for larger scale case studies to be possible. Stability and robustness concerns played a smaller role than foreseen in the DoW: Self-stabilization properties implicitly play an important role in the analysis of ABS-NET. However, stability and robustness becomes more central once attention moves beyond performance to security and fault tolerance.

1.5 List of Papers Comprising Deliverable D3.5

This section lists all the papers that comprise this deliverable, indicating how each paper is related to the main text of this deliverable. The papers have not yet been submitted for publication. As requested by the reviewers, the papers are not directly attached to Deliverable 3.5. A version of this deliverable with the papers attached is available on the HATS web site at the following url:


Paper 1: Location Independent Routing in Process Network Overlays

This paper [24] introduces the microABS-NET fragment of core ABS and its network semantics using location independent routing, and proves correctness of the network semantics in the sense of barbed equivalence with respect to a semantics given in standard rewriting logic style.

The paper is written by Mads Dam and a conference version is to be submitted during the late winter 2013. (Download Paper 1.)

Paper 2: Efficient and Fully Abstract Routing of Futures in Object Network Overlays

This paper [26] extends the results of the previous paper by considering a richer fragment, milliABS, of core ABS which includes futures, and extends the correctness results of [24] accordingly.

The paper is written by Mads Dam and Karl Palmskog and a conference version is to be submitted during the late winter 2013 (Download Paper 2.)

Paper 3: ABS-NET: Fully Decentralized Runtime Adaptation for Distributed Objects

The paper [25] extends the ABS-NET model to allow interesting test programs in ABS to be implemented, and examines runtime performance adaptation on top of a distributed simulator developed for the ABS-NET platform.

The paper is written by Mads Dam, Ali Jafari, Andreas Lundblad, and Karl Palmskog and a conference version is to be submitted during the late winter/early spring 2013 (Download Paper 3.)
Chapter 2
Location Independent Routing for Process Networks

2.1 Introduction

The decoupling of applications from their physical realization, intimately tied to the concept of virtualization, is a recurrent theme in the history of computing. Running applications on virtual machines allows many tasks to be performed independently of the physical computing infrastructure. By migrating virtual machine images between physical processors it is possible for a cloud provider to adapt processing and communication load to changing application demands and to changes in the physical infrastructure. In this way applications can, at least in principle, be freed of the burden of resource allocation. That is, it is left to an underlying processing network to determine on which nodes to place which tasks in order to make efficient use of processing resources, while at the same time meeting requirements on response times and processing capacity. If realized, the result is simpler application logic, better service quality, and, ultimately, lower costs for development, operations and management.

The question is how to realize this potential with a minimum of overhead, and in such a way that applications behave in a predictable manner. We examine this question in the context of a rudimentary distributed object language, and propose a formal, networked, runtime semantics of this language with some quite novel features. The goal is a “bare-metal” style of semantics where all aspects of computing and communication are accounted for in terms of local operations that could be directly implemented on top of silicon or, say, a hypervisor such as Xen [9].

A key problem is how to handle object and task mobility in an efficient manner. Since the allocation of objects to nodes is dynamic, some form of application level routing is needed to ensure that messages reach their destinations quickly, and with minimal overhead. Various approaches have been considered in the literature (cf. [78] for a survey):

- One option is to maintain a centralized or distributed database of object locations. Such a database can be used for both forwarding, by routing messages through the forwarding server, and for location querying, by using the database to look up destination object locations. In either case, object location and the location database must be kept consistent, which requires synchronization. Many experimental object mobility systems in the literature use some form of replicated or distributed location databases, cf. [35, 78, 12, 46].

- Another option is for nodes to maintain forwarding pointers, as in the Emerald system [60]. Migration then causes forwarding pointer chains to be extended by one further hop, and some mechanism is typically used to piggyback location update information onto messages, to ameliorate forwarding chain growth. This mechanism is used, for instance, in JoCaml [22].

- Many solutions involve some form of broadcast or multicast search. For instance, an object may use multicasting to find an object if a pointer for some reason has become stale, as in Emerald, or for
service discovery, as in Jini [4].

- Other solutions have been explored too, such as tree-structured DNS-like location directories [84], Awerbuch and Peleg’s distributed directories [7], and Demmer and Herlihy’s arrow protocol [34].

The main source of the difficulties these approaches are designed to solve is the distinction between destination host identifiers (location) and search identifiers (object identity). But, in a fully mobile setting the location at which an object resides has no intrinsic interest. What is of interest is message destination, that an RPC destined for the object with identity o is routed to the location where o resides, and not somewhere else. The address of the destination is not relevant. In other words, rather than routing messages according to IP address, inter object message routing should really be done according to the identity of the destination object rather than an assumed host identity, which might, for all the sender knows, be completely out of date.

This problem is in fact well known in the networking community, and has been the subject of significant attention over the last decade. The idea is to replace the location-based routing of traditional IP networks with location independent schemes that route messages according to names, or content. Names can be flat, unstructured identifiers, as in [14], or they can encode some form of signed content identity, as in content centric networking [53]. The general goal is to devise routing schemes for flat name spaces that are compact, such that routing tables can be represented at each node using space sublinear in the size of the network, and such that path lengths, and hence message latencies, do not grow too far from the optimal. The latter requirement of low stretch, defined as the ratio of route length to shortest path length, precludes the use of both location registers and hierarchical IP-like naming schemes.

The main purpose of this chapter is to show that name-based routing offers a new space for solutions to the object mobility problem with some attractive properties:

- No centralized or decentralized object location database is needed, since the network routing mechanism itself ensures that messages are routed to their proper hosts.

- A whole swathe of software becomes superfluous, which manages address lookups, message forwarding, rerouting, address bookkeeping, and the synchronization overhead between location registers and the migrating objects is eliminated. As a result, the “trusted computing base” of the networked execution platform is significantly reduced, in terms of size, and complexity.

- Traffic overhead is decreased. First, mobility support on top of IP needs to perform routing both at IP and at application level. Name-based routing in effect eliminates the need for IP level routing. Moreover, in steady state the simple distance vector routing scheme used here has stretch 1, so message delivery overhead is minimal (however, distance vector routing is not compact so our scheme does not scale well. We leave this issue for future work).

- Improved robustness: In faulty situations, if connection to the location register is lost, message delivery is impossible (or needs to resort to more costly mechanisms such as broadcasting, as in Emerald or Jini). Routing can be made self-stabilizing and thus able to adapt to any type of disturbance, as long as connectivity is maintained. This allows computation to progress (including delivery of messages and migration of objects) even when the network is under considerable churn.

On the other hand, throwing away network layers 3 and above may seem an excessive price to pay, and the argument above that the cost of IP routing should be taken into account is evidently invalid if IP routing needs to be performed anyway in any realistic implementation. We argue, however, that this does not need to be the case. First, as we have noted, the general architecture of the future internet is currently very much in flux. Second, although we have so far only explored an early prototype simulator [25] built on top of an IP-based overlay, it is perfectly possible today to build large scale non-IP networks with only layer 2

1 Location has interest as a source of latency, for instance, but that is another matter.

2 Routing tables in distance vector routing have size $\Omega(n \log n)$
connectivity, sufficient to bootstrap our approach. Third, the simplicity of our approach in comparison to the
task of formally verifying, e.g. IP and TCP \[13\], opens up for the possibility of extending currently ongoing
work on formally verified low level software along the lines of sel4 \[61\] to fully networked operating systems
and hypervisors at the device and instruction level. Finally, even if amending the current IP naming schemes
turns out to be infeasible, it is still of interest to evaluate the consequences of a much tighter integration
between network infrastructure and application level runtimes than what is currently done.

Our study is set in the context of a distributed object language microABS, a small fragment of core ABS
\[56\]. The microABS language includes only a rudimentary set of features, sufficient, however, to allow simple
networked programs to be programmed in a natural way, as we illustrate by a couple of small examples. The microABS
language is class based, and has features for remote method invocation, object creation, and
standard sequential control structures, similar to Sewell et al.’s Nomadic Pict language \[78\]. Two semantics
of microABS are given, one a standard semantics in rewriting logic style which does not take into account
aspects related to location, naming, routing, or communication. The second semantics takes these aspects
into account by executing objects on top of an arbitrary, but concrete, processor network. The main result
of the chapter is to show that the network semantics is sound and fully abstract with respect to the reference
semantics. We base the analysis on barbed equivalence \[73\]. Barbed equivalence requires a witness relation
that preserves some set of primitive observations, here remote calls to external objects, in both directions,
and is preserved under weak reductions, also in both directions. Barbed equivalence is convenient since the
required (unlabelled) reduction relations are easy and fairly uncontroversial to define, both for the reference
semantics and for the network semantics. Using a labelled transition semantics particularly at the network
level is much more complex, and needs to be subject to a separate justification which is out of scope for the
present work.

We structure the presentation as follows: In Section 2.2 we introduce the microABS language, briefly
discuss the reference semantics, and introduce the notion of barbed equivalence. The detailed semantics
follows the rewriting style of semantics used in many ABS-related works and can be found in \[24\]. We then
turn to the network semantics. In Section 2.3 we first introduce runtime states, or configurations, including
routing and network graphs, and the reduction relation. Details concerning well-formedness conditions that
need to apply at both the reference (so-called type 1) level and at the network (type 2) level are deferred
to the full version. In Section 2.4 we then summarize the soundness and full abstraction proof showing that
initial configurations at type 1 and type 2 levels are barbed equivalent. Finally, in Section 2.5 we conclude.

2.2 microABS

The syntax of the object language microABS is presented in Figure 2.1. We use a standard boldface notation

<table>
<thead>
<tr>
<th>x, y ∈ Var</th>
<th>Variables</th>
<th>e ∈ Exp</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>P ::= CL{x, s}</td>
<td>Program</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CL ::= class C(x){y, M}</td>
<td>Class definition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M ::= m(x){y, s}</td>
<td>Method definition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s ::= s1; s2</td>
<td>x = rhs</td>
<td>skip</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if e{s1} else {s2}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>while e{s}</td>
<td>!m(e)</td>
<td></td>
</tr>
<tr>
<td>rhs ::= e</td>
<td>new C(e)</td>
<td>Right hand sides</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.1: microABS abstract syntax

for vectors. Thus, e.g. x is a vector of variables. Programs are sequences of class definitions, along with
global variables x, and a "main" statement s. The class hierarchy is flat and fixed. Objects have parameters
x, local variable declarations y, and methods M. Methods have parameters x, local variable declarations y
and a statement body. For simplicity we assume that variables have unique declarations. The definition of
expressions e is left open, but we require that expressions are side-effect free. The language is untyped. It is
class Server1() {,
  serve(from, x) {, from!response(foo(x)) }
},
class Client(arg) {,
  use(server) {, server!serve(self, arg) }
  response(y) {, env!output(y) }
}

server, client,
server = new Server1() ;
client = new Client(42) ;
client!use(server)

Figure 2.2: Simple server

possible to add types and a notion of well-typedness. However, this does not affect the presentation in any significant way.

Besides standard sequential control structures (the choice of which is largely irrelevant), statements involve a minimal set of constructs for asynchronous method invocation and object creation. Method bodies lack a return statement. All interobject communication is therefore by RPC. Return statements with futures are considered in Chapter 3 of this deliverable. For now, method bodies are simply evaluated to the end at which point the evaluating task is terminated. In the absence of return statements, objects communicate using callbacks in a manner which is not dissimilar to communication in Erlang, as illustrated in the following examples.

Example 2.2.1. A very simple server applying foo to its argument is shown in Figure 2.2. We assume a reserved OID env with reserved method output to be used as a standard output channel.

Example 2.2.2. Just to show that the language is not trivial, the program in Figure 2.3 constructs an object ring with (here) 42 elements. A value is circulated along the ring, computing \( \text{bar}(\ldots(\text{bar}(x, 42), 41) \ldots, 1) \). Each ring element decrements an iterator \( \text{iter} \) initialized to the original value 42 first received. The final ring element returns the final value to the server, which then finally returns it to the client through the output method of object env.

We give a reduction semantics, the reference, type 1 semantics, for microABS in in terms of a transition relation \( cn \rightarrow cn' \) where \( cn, cn' \) are configurations, as determined by the runtime syntax in Figure 2.4. For a detailed explanation of the runtime syntax we refer to [24] and highlight only the main points here. Object identifiers, in particular, are names, and the \( \pi \)-like binder bind is used to bind OIDs. The binder acts as name binding in \( \pi \)-calculus, with similar scope intrusion and extrusion properties. For computations to have observable effects we assume a fixed set \( \text{Ext} \) of external OIDs with dedicated methods, such as the OID env and the method output in Examples 2.2.1 and 2.2.2. Call containers play a special, somewhat subtle role in defining the external observations of a configuration \( cn \). An observation, or \( \text{barb} \), is an external method call, i.e. a method call to an OID in \( \text{Ext} \). Calls that are not external are meant to be completed in usual reduction semantics style, by internal reaction with the called object, to spawn a new task. External calls could be represented directly, without introducing a special container type (which is not present in the core ABS semantics of [50]), by saying that a configuration \( cn \) has \( \text{barb} \) \( obs = o!m(v) \) if and only if \( cn \) has the shape

\[
\text{bind } o_1.(cn' \ o(o_2, a) \ t(o_2, l, e_1!m(e_2); s)) ,
\]

where \( e_1(a, l) = o \in \text{Ext} \) (the value of expression \( e \) in object environment \( a \) and method environment \( l \)) and \( e_2(a, l) = v \). However, in a semantics with unordered communication, which is what is assumed of
```java
class Server(){
    serve(from,x){
        c = new Cell();
        c!process(x, self, x)
    }
    return(result){
        from!response(result)
    }
},
class Cell(){
    process(x, root, iter){
        if iter = 0 {
            root!return(y)
        } else {
            c = new Cell();
            c!process(bar(x, iter), root, iter-1)
        }
    }
},
class Client(arg){
    use(server){
        server!serve(self, arg)
    }
    response(y){
        env!output(y)
    }
}
server, client,
server = new Server();
client = new Client(42);
client!use(server)
```

**Figure 2.3: Dynamic ring**

**Figure 2.4: microABS type 1 runtime syntax**

- $x \in \text{Var}$: Variables
- $o \in \text{OID}$: Object id
- $p \in PVal$: Primitive values
- $v \in Val = PVal \cup OID$: Values
- $l \in MEnv = \text{Var} \rightarrow Val$: Method environment
- $a \in OEnv = \text{Var} \rightarrow Val$: Object environment
- $tsk \in Tsk ::= t(o,l,s)$: Task
- $obj \in Obj ::= o(o,a)$: Object
- $cl \in Call ::= c(o,o',m,v)$: External call
- $ct \in Cnt ::= tsk \mid \text{obj}$: Container
- $cn \in Cn ::= 0 \mid ct \mid cn cn' \mid \text{bind } o.cn$: Configuration

core ABS \[56\], and which we also implement here, consecutive calls should commute, i.e., there should be no observational distinction between the method bodies $e_1!m_1(e'_1); e_2!m_2(e'_2)$ and $e_2!m_2(e'_2); e_1!m_1(e'_1)$. This, however, is difficult to reconcile with the representation \[2.1\]. To this end call containers are introduced, to allow configurations like \[2.1\] to produce a corresponding call, and then proceed to elaborate $s$.

In Figure 2.5 we give the mostly routine rewrite rules for contextual reasoning, and for the basic control structures. The remaining rewrite rules, for calls and object creation, are shown in Figure 2.6. We leave out of this summary the definitions of type 1 initial configuration, of type 1 reachability, and of type 1 well-formedness. The latter serves to ensure basic properties such as existence of objects with given OIDs, that tasks have corresponding objects, that no objects exists with an external OID, etc.

**Barbed Equivalence** Our approach to implementation correctness is based on the notion of barbed equivalence \[73\], a notion of equivalence often used to relate transition systems determined by a reduction semantics, cf. \[19, 10, 43\]. Our goal is to show that it is possible to remain strongly faithful to the reference semantics, provided all nondeterminism is deferred to be handled by a separate scheduler. This allows to
new semantics level we consider at present, and at the network level. For this reason the present account based
semantics introduced later, and such that the notions of context correspond at both the abstract, reference
It is, however, far from trivial to devise a natural notion of context that works at the level of the network
call. The derived predicate $cn \downarrow obs$ holds just in case $cn \rightarrow^* cn' \downarrow obs$ for some $cn'$.

Figure 2.5: milliABS reduction semantics part 1

Figure 2.6: Sample microABS reduction rules

draw strong conclusions also in the case a scheduler is added, as we discuss later. Barbed equivalence requires
of a pair of equivalent configurations that the internal transition relation $\rightarrow$ is preserved in both directions,
while preserving also a set of external observations. Although weaker than corresponding equivalences such
as bisimulation equivalence on labelled transition systems, barbed equivalence is nonetheless of interest for
the following two reasons:

1. Barbed equivalence offers a reasonable account of observationally identical behaviour on closed systems,
i.e., when composition of (in our case) subconfigurations to build larger configurations is not considered
because it a) is for some reason not important or relevant, or b) does not offer new observational
capabilities.

2. Barbed equivalence can be strengthened in a natural way to contextual equivalence\cite{71} by adding a
natural requirement of closure under context composition. Furthermore, a number of works\cite{54,74}
have established very strong relations between contextual equivalence for reduction oriented semantics
and bisimulation/logical relation based equivalences for sequential and higher-order computational
models.

It is, however, far from trivial to devise a natural notion of context that works at the level of the network
semantics introduced later, and such that the notions of context correspond at both the abstract, reference
semantics level we consider at present, and at the network level. For this reason the present account based
on barbed equivalence is also a natural stepping stone towards a deeper study of the notion of context in
real-world—or at least not overly artificial—networked software systems.

Let $obs = o'!m(v)$. The observation predicate $cn \downarrow obs$ is defined to hold just in case $cn$ can be written
in the form

$$\text{bind } o \cdot (cn' \ c(o', m, v))$$.

CTXT-1: If $cn_1 \rightarrow cn_2$ then $cn \vdash cn_1 \rightarrow cn_2$

CTXT-2: If $cn_1 \rightarrow cn_2$ then bind $z.cn_1 \rightarrow$ bind $z.cn_2$

WLOCAL: If $x \in \text{dom}(l)$ then $t(o, l, x = e; s) \rightarrow t(o, l[\hat{e}(a, l)/x], s)$

WFIELD: If $x \in \text{dom}(a)$ then $o(a, a) \ t(o, l, x = e; s) \rightarrow o(a, a[\hat{e}(a, l)/x]) \ t(o, l, s)$

SKIP: $t(o, l, \text{skip}; s) \rightarrow t(o, l, s)$

IF-TRUE: If $\hat{e}(a, l) \neq 0$ then $t(o, l, \text{if } e\{s_1\} \text{ else } \{s_2\}; s) \rightarrow t(o, l, s_1; s)$

IF-FALSE: If $\hat{e}(a, l) = 0$ then $t(o, l, \text{if } e\{s_1\} \text{ else } \{s_2\}; s) \rightarrow t(o, l, s_2; s)$

WHILE-TRUE: If $\hat{e}(a, l) \neq 0$ then $t(o, l, \text{while } e\{s_1\}; s) \rightarrow t(o, l, s_1; \text{while } e\{s_1\}; s)$

WHILE-FALSE: If $\hat{e}(a, l) = 0$ then $t(o, l, \text{while } e\{s_1\}; s) \rightarrow t(o, l, s)$

---

\begin{equation}
\text{CALL: Let } o' = c_1(a, l) \text{ in } o(a, a) \ o(o', a') \vdash t(o, l, e_1!m(e_2); s) \rightarrow t(o, l, s) \ t(o', \text{locals}(o', m), \text{body}(o', m))
\end{equation}

\begin{equation}
\text{CALL-EXT: If } o' = c_1(a, l) \in \text{Ext then } o(a, a) \vdash t(o, l, e_1!m(e_2); s) \rightarrow t(o, l, s) \ c(o', o', m, e_2(a, l))
\end{equation}

\begin{equation}
\text{NEW: } o(a, a) \vdash t(o, l, x = \text{new } C(e); s) \rightarrow \text{bind } o'.t(o, l[o'/x], s) \ o(o', \text{init}(C, \hat{e}(a, l)))
\end{equation}
Let now $\mathcal{R}$ be a binary relation on type 1 well-formed configurations. We are interested in relations with the following properties:

- **Symmetry**: If $cn_1 \mathcal{R} cn_2$ then $cn_2 \mathcal{R} cn_1$
- **Reduction-closure**: If $cn_1 \mathcal{R} cn_2$ and $cn_1 \rightarrow cn'_1$ then there exists some $cn'_2$ such that $cn_2 \rightarrow^* cn'_2$ and $cn'_1 \mathcal{R} cn'_2$
- **Barb preservation**: If $cn_1 \mathcal{R} cn_2$ and $cn_1 \downarrow obs$ then $cn_2 \downarrow obs$

We call a relation with these three properties a *type 1 witness relation*.

**Definition 2.2.3** (Type 1 Barbed Equivalence). Let $cn_1 \simeq_1 cn_2$ if, and only if, $cn_1 \mathcal{R} cn_2$ for some type 1 witness relation $\mathcal{R}$.

### 2.3 Network Semantics: Runtime Configurations

The “standard” (type 1) semantics for microABS is quite abstract and does not account for many issues which must be faced by an actual implementation, in particular if the goal is high performance and scalability. For instance:

- The microABS semantics implements a rendez-vous oriented communication model. We want to account for this using a standard buffered asynchronous model.
- Accordingly, calls should be replaced by message passing.
- The microABS semantics has no concept of proximity or name space. Any two objects, regardless of their “location” can without any overhead or search choose to synchronize at any point. Instead, we want a semantics that is *network aware* in the sense that it brings out proximity and location without unduly constraining the model, for instance to a particular naming discipline, or to a centralized name or location lookup service.

Our proposal is to execute microABS objects on a network graph in a fully decentralized and lock free manner where the only means of communication or synchronization is by asynchronous message passing along edges connecting neighbouring nodes, each edge having an associated directional, buffered communication channel. In this section we accordingly introduce a refinement of the standard semantics, a “network semantics”, or type 2 semantics, which adds an explicit network components to the type 1 semantics. The key idea is to use name-based routing, as explained in the introduction. That is, nodes are equipped with explicit routing information allowing messages to be addressed to specific receiving objects, rather than their hosts, which may change. This allows a very simple, fully decentralized, and lock free integration of routing and object migration, as we now begin to demonstrate.

#### Nodes and Routing

The network semantics is presented in rewriting logic style, similar to the type 1 semantics above. We still have configurations $cn$, but these now have a richer structure. We first introduce two new types of container to reflect the underlying network graph, namely *nodes* and *links*. Node containers have the form $n(u,t)$ where $u \in NID$ is a primitive *node identifier*, and where $t$ is an associated routing table. Node identifiers (NIDs) take the place of IP addresses in the usual IP infrastructure. For routing we assume a rudimentary Bellman-Ford distance vector routing discipline [82]. More elaborate and practical routing schemes exist that are better equipped for e.g. disconnected operation, and with better combinations of scalability and stretch. However, for the present purpose, the distance vector scheme is quite adequate. Consequently, a routing table $t$ is a partial function associating to the OIDs $o$ “known” to $t$ a pair $t(o) = (u,n)$ where $n$ is the minimum number of hops believed by $t$ to be needed to reach the node hosting $o$ from the current node, and where $u$ is the next hop destination.
Routing Tables  Routing tables support the following operations:

- Next hop lookup, \( \text{nxt}(o, t) = \pi_1(t(o)) \): In the context of a node \( n(u, t) \), \( \text{nxt}(o, t) \) returns a neighbour \( u' \) of \( u \) to which, according to the current state of \( u \), a message should be sent in order to eventually reach the destination \( o \).

- Update, \( \text{upd}(t, u, t') \): Updates \( t \) by incorporating the routing table \( t' \) belonging to a (neighbouring) node \( u \). The update function is defined thus:

\[
\text{upd}(t, u, t')(o) = \begin{cases} 
\bot & \text{if } o \notin \text{dom}(t) \cup \text{dom}(t') \\
\text{t}(o) & \text{else, if } o \notin \text{dom}(t') \\
(u, \pi_2(t'(o)) + 1) & \text{else, if either } o \notin \text{dom}(t), \text{ or } \pi_1(t'(o)) = u, \\
t(o) & \text{or } t'(o) < \pi_2(t(o)) - 1 \\
& \text{otherwise}
\end{cases}
\]

If \( o \) is known to neither the current node nor to \( u \), the distance estimate to \( o \) from the current node is undefined. If it is known to the current node, but not to \( u \), \( u \)'s information is unchanged. If it is known to \( u \), but not to the current node, the estimate from the current node becomes 1 plus \( u \)'s estimate. Otherwise we may assume that \( u \) is known to both the current node and to \( u \). If the minimal route follows the edge between the current node and \( u \), \( u \)'s distance estimate plus one is the new distance estimate at the current node, regardless of whether the estimate improves on the current estimate or not. Otherwise, if \( u \)'s estimate improves sufficiently on the estimate at the current node, \( u \)'s estimate is incremented and used at the current node. In other circumstances, the distance estimate at the current node is left unchanged.

- Registration, \( \text{reg}(o, u, t) \): Returns the routing table \( t' \) obtained by registering \( o \) at \( u \) in \( t \), i.e.

\[
\text{reg}(o, u, t)(o') = \begin{cases} 
(u, 0) & \text{if } o = o' \\
t(o') & \text{otherwise}
\end{cases}
\]

The function \( \text{reg} \) is invoked only when \( u \) is the “current” node.

Links, Queues, and Messages  Nodes are connected by directed edges, or links, of the form

\[ l(u, q, u') \]

where \( u \in \text{NID} \) is the source NID, \( u' \in \text{NID} \) is the sink NID, and where \( q \in Q \) is the associated fifo message queue. Queue operations are standard: \( \text{enq}(msg, q) \) enqueues the message \( msg \) onto the tail of \( q \); \( \text{hd}(q) \) returns the head of \( q \), and \( \text{deq}(q) \) returns the tail of the \( q \), i.e., \( q \) with \( \text{hd}(q) \) removed. If \( q \) is empty (\( q = \varepsilon \)) then \( \text{hd}(q) \) and \( \text{deq}(q) \) are both undefined.

Messages have one of the following three forms:

- \( \text{call}(o, o', m, v) \): A remote call message originating from object \( o \) and addressed to object \( o' \), of method \( m \), and with arguments \( v \).

- \( \text{table}(t) \): A routing table update message. The origin NID is implicit, as the message is dequeued from a link queue with explicit source NID.

- \( \text{object}(cn) \): An object migration message, where \( cn \) is an object closure, as explained below.

Call messages are said to be object bound, and table and object messages are said to be node bound. We define \( \text{dst}(msg) \), the destination of \( msg \) to be \( o' \) for call messages, and \( \text{dst}(msg) = \bot \) in the remaining two cases.
The Network Graph  Nodes and links induce a directed graph structure \( graph(cn) \) in the obvious way, by taking as vertices the NIDs \( u \) and as edges pairs \( (u, u') \) for each link \( l(u, q, u') \). For this to make sense we impose some constraints that apply, from now on, to all “global” configurations \( cn \) in the type 2 semantics.

1. Unique vertices: There is at most one container \( n(u', t) \in cn \) with \( u' = u \).
2. Unique edges: For each source-sink pair \( u, u' \) there is at most one link \( l(u, q, u') \), for some \( q \).
3. Edges connect vertices: If \( l(u, q, u') \in cn \) then \( n(u, t), n(u', t') \in cn \) for some \( t, t' \).
4. Reflexivity: \( graph(cn) \) is reflexive, i.e., if \( n(u, t) \in cn \) for some \( t \) then \( l(u, q, u) \in cn \) for some \( q \).
5. Symmetry: \( graph(cn) \) is symmetric, i.e., if \( l(u, q, u') \in cn \) then \( l(u', q', u) \in cn \) for some \( q' \).
6. Connectedness: \( graph(cn) \) is connected, i.e., if \( n(u, t), n(u', t') \in cn \) then there is a path in \( graph(cn) \) connecting vertices \( u \) and \( u' \).

Condition 1 is essential for naming. Condition 2 is important, as we focus on closed systems. Condition 3 simplifies communication but could be lifted in principle. Conditions 4 and 5 are non-essential, but helpful. Finally, Condition 6 is essential for routing to stabilize, but many of the results below can be proved without it.

Objects and Tasks  In the type 2 semantics object containers are now attached to a node \( u \) and have the shape

\[
o(o, a, u, q_{in}, q_{out})
\]

where \( o \in OID \), \( a \in OEnv \) as before, and \( q_{in}, q_{out} \) is a pair of an ingoing and an outgoing fifo message queue. This object level buffering is not essential, as messages are already buffered at link level, but object level buffering allows a more elegant formalization. It is commonplace in actor languages to consider inbound queues only. Here we find it more elegant to allow an outgoing queue as well, although this is mainly a matter of taste. Tasks \( t(o, l, s) \) are unchanged from the type 1 semantics.

Object Closures  Type 2 configurations are built from the four container types introduced above, nodes, links, objects and tasks. It remains to explain object closures. For an object message \( OBJECT(cn) \) to be valid, the configuration \( cn \) needs to be an object closure of the form

\[
o(o, a, u, q_{in}, q_{out}) \ t(o, l_1, s_1) \ldots t(o, l_n, s_n)
\]

Specifically, if \( cn \) is any configuration then \( clo(cn, o) \), the closure of object \( o \) with respect to \( cn \), is the multiset of all type 2 containers of the form either \( o(o', a', u', q'_{in}, q'_{out}) \) or \( t(o', l', s') \) such that \( o' = o \), and \( objof(cn) \) is a partial function returning \( o \) if all type 2 containers in \( cn \) are either objects or tasks, with OID \( o \).

Type 2 Runtime Syntax  Reflecting the above description, the type 2 runtime syntax is presented in Figure 2.7. A pictorial representation of the type 2 runtime state is shown in Figure 2.8. Configurations remain multisets, and we write, e.g., \( obj \in cn \) if \( cn \) can be written as \( obj \ cn' \) for some \( cn' \). Tasks are unchanged from Figure 2.4. We write \( t(cn) \) for the multiset of tasks in \( cn \), i.e., the multiset \( \{ tsk \ | \ \exists cn'.cn = tsk.cn' \} \), and \( o(cn) \) for the multiset of objects in \( cn \), similarly defined. We use \( \preceq \) for the subterm relation, and write \( m(cn) \) for the multiset \( \{ msg \ | \ msg \preceq cn \} \). To avoid explosion of the notation we reuse symbols from the type 1 semantics as far as possible, and resolve them by context.

17
Type 2 Reductions  An important distinction between the standard semantics and the network semantics is the absence of binding. For the standard semantics, name binding plays a key role to avoid clashes between locally generated names. However, in a language with NIDs this device is no longer needed, as globally unique names can be guaranteed easily by augmenting names with their generating NID. Since all name generation takes place in the context of a given NID, we can simply assume an operation newo(u) that returns a new OID, which is globally fresh for the “current configuration”. Another important point to note is that all transitions in the type 2 semantics are fully local, in the sense that all operations applied, and all conditions determining whether or not a transition is enabled, can be fully determined by inspecting only one node and, possibly, the head of incoming link queues, alternatively by enqueuing messages to the tail of the outgoing queue.

A number of reduction rules, for instance for most of the program constructs, are common to the type 1 and type 2 semantics, and deferred to [21]. The most interesting reduction rules are presented in Figure 2.9. The rules are naturally divided into subgroups:

- The rules t-send and t-rcv are concerned with the exchange of routing tables.
- The three rules msg-send, msg-rcv and msg-route are used to manage message passing, i.e., reading a message from a link queue and transferring it to the appropriate object in-queue, and dually, reading a message from an out-queue and transferring it to the attached link queue. Finally, messages are routed to the next link, if the destination object does not reside at the current node. In rule msg-rcv
T-SEND:  \( n(u,t) \vdash l(u,q,u') \rightarrow l(u,\text{enq}(\text{TABLE}(t),q),u') \)

T-RCV:  If \( \text{nxt}(q) = \text{TABLE}(t') \) then \( l(u',q,u) \ n(n(u,t) \rightarrow l(u',\text{deq}(q),u) \ n(n(u,\text{upd}(t,u',t'))) \)

MSG-SEND:  If \( \text{hd}(\text{qout}) = \text{msg}, \text{dst}(\text{msg}) = o' \) and \( \text{nxt}(o',t) = u' \) then
\[
\begin{align*}
n(n(u,t) \vdash l(u,q,u) \ o(o,a,u,qin,qout) \rightarrow l(u,\text{enq}(\text{msg},q),u') \ o(o,a,u,qin,\text{deq}(\text{qout}))
\end{align*}
\]

MSG-RCV:  If \( \text{hd}(q) = \text{msg} \) and \( \text{dst}(\text{msg}) = o \) then
\[
\begin{align*}
l(u',q,u) \ o(o,a,u,qin,qout) \rightarrow l(u',\text{deq}(q),u) \ l(u,\text{enq}(\text{msg},q'),u')
\end{align*}
\]

MSG-Delay-1:  If \( \text{hd}(q) = \text{msg}, \text{dst}(\text{msg}) = o \) and \( \text{nxt}(o,t) = u'' \neq u \) then
\[
\begin{align*}
n(n(u,t) \vdash l(u',q,u) \ l(u',q',u'') \rightarrow l(u',\text{deq}(q),u) \ l(u,\text{enq}(\text{msg},q'),u')
\end{align*}
\]

MSG-Delay-2:  If \( \text{hd}(\text{qout}) = \text{msg}, \text{dst}(\text{msg}) = o' \) and \( \text{nxt}(o',t) \uparrow \) then
\[
\begin{align*}
n(n(u,t) \vdash l(u',q,u) \ l(u',q',u) \rightarrow l(u',\text{deq}(q),u) \ l(u,\text{enq}(\text{msg},q'),u)
\end{align*}
\]

CALL-SEND:  Let \( o' = e_1(a,f) \), \( \mathbf{v} = e_2(a,l) \) in
\[
\begin{align*}
&\ o(o,a,u,qin,qout) \quad t(o,l,e_1\text{m}(e_2);s) \rightarrow \\
&\ o(o,a,u,qin,\text{enq}(\text{CALL}(o',m,\mathbf{v}),\text{qout})) \ t(o,l,s)
\end{align*}
\]

CALL-RCV:  If \( \text{hd}(\text{qin}) = \text{CALL}(o',m,\mathbf{v}) \) then
\[
\begin{align*}
o(o,a,u,qin,qout) \rightarrow o(o,a,u,\text{deq}(\text{qin}),\text{qout}) \ t(o,\text{locals}(o,m,\mathbf{v}),\text{body}(o,m))
\end{align*}
\]

NEW-2:  Let \( o' = \text{new}(u) \) in
\[
\begin{align*}
o(o,a,u,qin,qout) + l(n(u,t) \ t(o,l,x = \text{new}(C(\mathbf{v}));s) \rightarrow \\
n(n(u,\text{reg}(o',u,t)) \ t(o,l[o'/x],s) \ o(o',\text{init}(C,\mathbf{e}(a,l)),u,v,\mathbf{e}))
\end{align*}
\]

OBJ-SEND:  Let \( \text{cn}' = \text{clo}(\text{cn},o) \) in
\[
\begin{align*}
n(n(u,t) \ l(u,q,u') \ \text{cn} \rightarrow n(n(u,\text{reg}(o,u',t))) \ l(u,\text{enq}(\text{OBJECT}(\text{cn'}),q),u') \ (\text{cn} - \text{cn'})
\end{align*}
\]

OBJ-RCV:  If \( \text{hd}(q) = \text{OBJECT}(\text{cn'}) \) then
\[
\begin{align*}
l(u',q,u) \ n(n(u,t) \rightarrow l(u',\text{deq}(q),u) \ n(n(u,\text{reg}(\text{objof}(\text{cn'},u),t))) \ \text{cn'}
\end{align*}
\]

Figure 2.9: Type 2 reduction rules

Note that the receiving node is not required to be present. This, however, will be enforced by the well-formedness condition later.

- In the rules MSG-Delay-1 and MSG-Delay-2 we use the notation \( \text{nxt}(o,t) \uparrow \) to denote the condition that \( \text{nxt}(o,t) \) is undefined. These rules are used to handle the case where routing tables have not yet stabilized. For instance it may happen that updates to the routing tables have not yet caught up with object migration. In this case, a message may enter an out-queue without the hosting node’s routing table having information about the message’s destination (rule MSG-Delay-2). Another case is where a node receives a message on a link without knowing where to forward it (rule MSG-Delay-1). This situation is particularly problematic as a blocked message may prevent routing table updates to reach the hosting node, thus causing deadlock. The solution we propose is to use the network self-loop as a buffer for temporarily unroutable messages.

- The rules CALL-SEND and CALL-RCV produce and consume call messages in a pretty obvious way.

- The rule NEW-2 handles object creation, including registration of the new object at the local node.

- The final two rules concern object migration. Of these, OBJ-SEND is a global rule in that it is not allowed to be used in subsequent applications of the CTXT-1 rule. In this way we can guarantee that only complete object closures are migrated. In rule OBJ-SEND, \( \text{cn} - \text{cn'} \) is multiset difference.

We emphasize again that all of the above rules are strictly local and appeal only to mechanisms directly implementable at link level: Tests and simple datatype manipulations take place at a single node, or accesses
the nodes link layer interface. The “global” property appealed to above for the migration rules is merely a formal device to enable an elegant treatment of object closures.

The reduction rules can be optimized in several ways. For instance, object self-calls are always routed through the ‘network interface’, i.e., the hosting node’s self-loop. This is not necessary. It would be possible to add a rule to directly spawn a handling task from a self call without affecting the results.

In [24] we introduce a notion of well-formedness at type 2 level, which we leave out of this presentation.

**Type 2 Barbed Equivalence**  We next adapt the notion of barbed equivalence to the type 2 setting. The only difficulty is to define the type 2 correlate of the observation predicate. We take the point of view that an observation \( \text{obs} = o!m(v) \) is enabled at a configuration \( cn \) if a corresponding call message \( \text{call}(o', o, m, v) \) is located at the head of one of the object output queues in \( cn \). More precisely, the type 2 observability predicate is \( cn \downarrow \text{obs} \), holding if and only if \( cn \) has the following shape;

\[
 cn = cn' \circ(o', a, u, q_{in}, q_{out}) \quad (2.2)
\]

and \( hd(q_{out}) \) is defined and equal to \( \text{call}(o', o, m, v) \).

There are other ways of defining the observability predicate that may be more natural. For instance one may attach external OIDs to specific NIDs and restrict observations to those NIDs accordingly. It is also possible to add dedicated output channels to the model, and route external calls to those. None of these design choices have any effect on the subsequent results, however, but add significant notational overhead, particular in the latter case.

With the observation predicate set up, the weak observation predicate is derived as in Section 2.2, and, as there, we define a type 2 witness relation as a relation that satisfies symmetry, reduction closure, and barb preservation. Thus:

**Definition 2.3.1 (Type 2 Barbed Equivalence).** Let \( cn_1 \equiv_2 cn_2 \) if and only if \( cn_1 \mathcal{R} cn_2 \) for some type 2 witness relation \( \mathcal{R} \).

In fact, for our purpose there in no real need to distinguish between the type 1 and type 2 equivalences, and hence we conflate the notions of witness relation and barbed equivalences, by letting the type of the configuration arguments be determined by the context, and use \( \equiv \) as the generic notion.

### 2.4 Correctness

The goal is to prove that if \( cn_1 \) and \( cn_2 \) are initial type 1 and type 2 configurations, respectively, for the same program, then \( cn_1 \equiv cn_2 \). The key to the proof is a normal form lemma for the type 2 semantics saying, roughly, that any well-formed type 2 configuration can be rewritten, using a subset of the rules as detailed below, into a form where queues have been emptied of all routable messages, where routing tables have been in some expected sense normalized, and where all objects have been moved to a single node. We prove this in two steps. First we prove a stabilization result, that non-self links can be emptied of messages and routing tables normalized to induce messaging paths with unit stretch. This allows the second normalization step to empty also object queues and migrate all objects to a single node. Once this is done we can prove correctness by exhibiting a map representing each type 1 configuration as a canonical type 2 configuration, using normalization to help prove reduction preservation in both directions. Then only barb preservation is needed to complete the correctness argument.

**Stabilization**  We first show that each configuration can be rewritten using the transition rules into a form for which routing is stable and all queues are empty, except for external messages, i.e., messages \( msg \) addressed to an object \( o \in \text{Ext} \). By well-formedness we then know that no object \( o(o', a', u', q_{in}, q_{out}) \preceq cn \) with \( o' = o \) exists. In the context of a configuration \( cn \) call a proper link any link \( l(u, q, u') \) for which \( u \neq u' \).

**Definition 2.4.1 (Stable Routing, External Link Messages).** Let \( cn \) be a well-formed type 2 configuration.
Algorithm 1: Stabilize routing and read internal link messages

**Input** Type 2 well-formed configuration \( cn \) on a connected network graph

**Output** Configuration with stable routing and external link messages only

**repeat**

Use T-\textit{send} on each proper link in \( cn \) to broadcast routing tables to all neighbours;

**repeat**

Use T-\textit{rcv} to dequeue one message on a link in \( cn \) until T-\textit{rcv} no longer enabled;

Use MSG-\textit{rcv}, MSG-\textit{route}, MSG-\textit{delay-1}, OBJ-\textit{rcv} to dequeue one message from each link, if possible

**until** link queues contain only external messages, and routing is stable

Figure 2.10: Algorithm 1 – Stabilize routing and empty link queues of internal messages

1. \( cn \) has stable routing, if for all \( n(u,t), o(o,a,u',q_{in},q_{out}) \leq cn \), if \( \text{nxt}(o,t) = u'' \) then there is a minimal length path from \( u \) to \( u' \) which visits \( u'' \).

2. \( cn \) has external link messages only, if \( l(u,q,u') \in cn \) and \( msg \leq q \) implies \( u = u' \) and \( msg \) is external.

The strategy for performing the rewriting is to first empty link queues as far as possible as we simultaneously exchange routing tables to converge to a configuration with stable routing. This first stage is accomplished using Algorithm 1 in Figure 2.10 where we hide uses of CTXT-1 to allow the transition rules to be applied to arbitrary containers. Observe that we have no intention to use Algorithm 1 or any of the later algorithms in this section to do actual computing in the type 2 semantics. “Real” network computing using the type 2 semantics requires more sophisticated approaches. The algorithms considered here do not need to be effective or “local”: We only need to exhibit some strategy for producing a configuration with the desired result, allowing us to prove the desired normal form results.

**Proposition 2.4.2.** Algorithm 1 terminates.

Write \( A_1(cn) \Rightarrow cn' \) if the configuration \( cn' \) is a possible result of applying Algorithm 1 to \( cn \). We then say that \( cn' \) is in stable form. Stable forms are almost unique, but not quite, since routing may stabilize in different ways, and since this (plus the generally nondeterministic scheduling of rules in Algorithm 1) may cause messages to enter object input queues at different times. The main result, shown in [24] is that stabilization preserves barbed equivalence.

**Theorem 2.4.3.** If \( A_1(cn) \Rightarrow cn' \) then \( cn \simeq cn' \).

**Normalization** We then turn to the second normalization step, to empty object queues and migrate all object closures to a central node. The normalization procedure is Algorithm 2 shown in Figure 2.11. Let \( A_2(cn) \Rightarrow cn' \) if \( cn' \) is a possible result of applying Algorithm 2 to \( cn \). Initially a node \( u_0 \) is chosen towards which all objects will migrate during normalization. Normalization is performed in cycles, each cycle starting and ending in a stable configuration. In each cycle, first object in- and out-queues are emptied. Then, objects not yet at \( u_0 \) are migrated one step towards \( u_0 \). Routing is not needed for this. It is sufficient to know that migration toward \( u_0 \) is possible.

**Proposition 2.4.4.** Algorithm 2 terminates.

Through a somewhat elaborate normal form argument we can establish the following result:

**Theorem 2.4.5.** If \( A_2(cn) \Rightarrow cn' \) then \( cn \simeq cn' \).
Algorithm 2: Normalization

**Input** Type 2 well-formed configuration \( cn \) on a connected network graph

**Output** Configuration in type 2 normal form

fix a NID \( u \);
run Algorithm 1;
repeat
  while some object queue is nonempty {
    use MSG-SEND, MSG-DELAY-2, CALL-RCV to dequeue one message from each
    nonempty object queue } ;
  while an object exists not located at \( u \) {
    use OBJ-SEND to send the object towards \( u \) } ;
run Algorithm 1
until all objects are located at \( u \) and queues contain only external messages

Figure 2.11: Algorithm 2 – Normalization

**Correctness** The goal is to prove soundness and full abstraction of the network semantics, i.e., that for any two type 1 configurations \( \text{bind } o . cn, \text{bind } o' . cn' \) in standard form, \( \text{bind } o . cn \simeq \text{bind } o' . cn' \) if and only if \( \text{down}(cn_1) \simeq \text{down}(cn_2) \). However, since we have set up the semantics such that \( \simeq \) applies without modification at both type 1 and type 2 levels it suffices to prove that \( \text{bind } o . cn \simeq \text{down}(cn) \).

To accomplish this we represent each type 1 configuration as a type 2 configuration in normal form. We first fix an underlying graph represented as a well-formed type 2 configuration \( cn_{graph} \) and a distinguished UID \( v_0 \) in this graph, similar to the way initial configurations are defined in Section 2.3. Thus, \( cn_{graph} \) consists of nodes and links only, each node \( u \) in \( cn_{graph} \) has the form \((u, t)\), and each link has the form \((u, e, u')\). The routing tables \( t \) are defined later. Defining a suitable representation map is a little cumbersome. A first complication is that names in the type 1 semantics (which includes the binder) need to be related to names in the type 2 semantics, which does not include the binder, but on the other hand has different generator functions (the function \text{newo}). For external names this is not a problem, but for bound names some form of name representation map is useful to connect the two types of names. Accordingly, we fix an injective \text{name representation map} \( \text{rep} \), taking names \( o \) in the type 1 semantics to names \( \text{rep}(o) \) in the type 2 semantics. For convenience we extend the name representation map \( \text{rep} \) to external names \( o \in \text{Ext} \) by \( \text{rep}(o) = o \), to arbitrary values by \( \text{rep}(p) = p \), to task environments by \( \text{rep}(l)(x) = \text{rep}(l(x)) \) and similarly for object environments. The only slight complication in defining the mapping \( \text{down} \) is that we need an operation to send a type 1 call container as a message in the type 2 semantics. This is done by the operation \text{send} which sends a call container originating at \( o \) onto object \( o' \)'s output queue as follows:

\[
\text{send}(c(o, o', m, v), o(a, a, u, q_{in}, q_{out}) \ cn) = o(a, a, u, q_{in}, \text{enq}(\text{CALL}(o, o', m, v), q_{out})) \ cn \tag{2.3}
\]

We can then define the type 2 representation of the type 1 configuration \( \text{bind } o . cn \) (leaving routing tables to be defined shortly) as the extended configuration \( \text{down}(cn)(cn_{graph}) \) where \( \text{down} \) is defined by induction on the structure of \( cn \) as follows:

- \( \text{down}(0)(cn) = cn \).
- \( \text{down}(cn_1 \ cn_2) = \text{down}(cn_1) \circ \text{down}(cn_2) \).
- \( \text{down}(t(o, l, s))(cn) = t(\text{rep}(o), \text{rep}(l), s) \ cn \).
- \( \text{down}(o(o, a))(cn) = o(\text{rep}(o), \text{rep}(a), u_0, e, e) \ cn \).
- \( \text{down}(c(o, o', m, v))(cn) = \text{send}(c(\text{rep}(o), \text{rep}(o'), m, \text{rep}(v)), cn) \).
In other words, we represent type 1 configurations by first assuming some underlying network graph, and then mapping the containers individually to type 2 level. The only detail remaining to be fixed above is the routing tables. For \( u_0 \) the initial routing table, \( t_0 \), needs to register all objects in \( cn_0 \), i.e.,
\[
t_0 = \text{reg}(g(o_0), u_0, \text{reg}(g(o_1), u_0, \text{reg}(\cdots, \text{reg}(g(o_m), u_0, \bot)) \cdots ))
\]
where \( o_0, \ldots, o_m \) are the OIDs in \( cn_0' \). For nodes \( n(u, t) \) where \( u \neq u_0 \) we let \( t \) be determined by some stable routing. This is easily computed using Algorithm 1, and we leave out the details. This completes the definition of \( \downarrow(cn) \).

**Lemma 2.4.6.** Let \( \text{bind } z.cn \) be type 1 well-formed in standard form.

1. If \( \text{bind } z.cn \rightarrow \text{bind } z'.cn' \) then \( \downarrow(cn) \rightarrow^* \circ \simeq \downarrow(cn') \).

2. If \( \downarrow(cn) \rightarrow cn'' \) then for some \( z', cn', \text{bind } z.cn \rightarrow^* \text{bind } z'.cn' \) and \( cn'' \simeq \downarrow(cn') \) \( \square \).

We can now prove the main correctness result.

**Theorem 2.4.7 (microABS-NET Implementation Correctness).** For all well-formed type 1 configurations \( cn \) on connected network graphs, \( cn \simeq \downarrow(cn) \) \( \square \).

### 2.5 Discussion

We have presented a sound and fully abstract semantics for a rudimentary object language, in terms of a novel network-based execution model. Thanks in part to the explicit mixing of messaging and routing we are able to present the model at a level where it could in principle be implemented in a provably correct fashion directly on top of silicon, or integrated in a hypervisor such as Xen [9], assuming reliable link layer functionality only. This is a direction of work we aim to explore in the future.

Soundness and full abstraction is a useful validation that the network semantics induces the same behaviour on microABS programs as the reference semantics. The network semantics, however, lacks a scheduler to determine, e.g., when to migrate objects and how to schedule threads on single nodes. Such a scheduler will resolve nondeterministic choices left open in the network semantics presented here. Once such a scheduler is added, soundness and full abstraction is lost. As we show in the following chapter we can, however, easily adapt the results to a notion of barbed simulation, obtained by instead of requiring preservation of observations and reductions in both directions, requiring preservation only in one. Then correctness for barbed simulation, that \( \downarrow(cn) \) simulates \( \text{bind } o.cn \), is obtained as a corollary.

Substantial work has been going on in the HATS project on the ABS language [56] and its extensions, for instance towards software product lines [76]. Johnsen et al. [59] suggest an extension of ABS with deployment components for resource management. We are mainly interested in the microABS language as an example. Essentially, however, our work is language independent, and we could apply the approach presented here to a version of core Erlang with minor changes only. Some details would be different, in particular the treatment of Erlang’s pattern match-based message reception construct. The changes, however, would be local only, and so make little essential difference.

Much work has been done on object/component mobility in the \( \pi \)-calculus tradition [66], and on the implementation of high-level object or process-oriented languages in terms of more efficiently implementable low level calculi. In [78], following earlier work on Pict [70], Fournet’s distributed join-calculus [41], and the JoCaml programming language [22], a compiler is implemented and proved correct for Nomadic Pict, a prototype language with very similar functionality to our microABS language: principally asynchronous message passing between named, location-oblivious processes. The target language extends Pierce and Turner’s Pict language with synchronous local communication and asynchronous message passing between located processes. In comparison with [78] the use of name-based routing allows us to use barbed equivalence in place of coupled simulation [69] and as a consequence obtain a simpler correctness proof, due to the need for locking in the central forwarding server scheme used in [78]. JoCaml also uses forward chaining, along with an
elaborate mechanism to collapse the forwarding chains. In the Klaim project [12] compilers were implemented and proved correct for several variants of the Klaim language, using the Linda tuple space communication model and a centralized name server to identify local tuple servers. The Oz kernel language [81] uses a monotone shared constraint store in the style of concurrent constraint programming. The Oz/K language [63] adds to this a notion of locality with separate failure and mobility semantics, but no real distribution or communication semantics is given (long distance communication is reduced to explicit manipulation of located agents, in the style of Ambient calculus [16]).

Our correctness proof uses reduction semantics and barbed equivalence. This is rather standard in the process algebra literature, cf. [40, 19, 43]. Both Sewell et al. [78] and Fournet et al. [42] use coupled simulation in order to handle problems related to preemptive choice. This complication does not arise in our work, whence barbed equivalence suffices. However, barbed equivalence is mostly useful for closed systems modeling where the stimuli to which an observed system is to be exposed must be given up front, as part of the initial configuration. A structural account based on some form of contextual equivalence [71], or on bisimulation equivalence along with a labelled transition semantics instead, would be more suitable. Work in this direction is currently going on. Replacing our reference reduction-based semantics with a labelled transition semantics is fairly straightforward. The bigger challenge is to develop a suitable structural account at the network level, allowing partially defined configurations to be composed.

Another direction for future work is to extend the microABS language. Experiments in this directions are going on. In [26], reported in the next chapter, we extend microABS with futures (aka promises [64]) as placeholders for return values. This extension turns out to be not at all trivial. In other directions it is of interest to consider models with node power-on/power-off, in order to model systems with adaptive power consumption, as well as various forms of node failure, along similar lines as [37]. The model can be used as a platform for language-based studies of load balancing and resource adaptation. We have extended the network semantics reported here to the full core ABS language [56] and implemented a multi-core simulation engine. This work is reported in [25]. There we show how the network semantics presented here can be split into a language interpreter layer and a language independent node controller layer, not unlike the meta-actors of [62], and we show how different resource allocation heuristics can be used to optimize object-to-node placement for different simple applications.

Further down the line it is of interest to examine the potential practical implications of our approach. This presents additional challenges, including garbage collection and buffer management, in particular as the network semantics we have presented above uses unbounded buffers. Also the routing scheme must be reconsidered. Distance vector routing suffers from well-known and fundamental scalability and security problems, and needs consideration in light of recent progress on compact routing (cf. [83, 79]).
Chapter 3

Efficient Routing of Futures in Object Networks

3.1 Introduction

In [24], summarized in Chapter 2 of this deliverable, we introduce a novel model for highly adaptable and efficient networked execution of object-based programs. The key idea is to use a form of location independent, or flat, routing [51, 14, 53] that allows messages (RPCs) to be routed directly to the called object, independent of the physical node on which that object is currently executing. In this way a lot of the overhead and performance constraints associated with object mobility can be eliminated, including latency and bandwidth overhead due to looking up, querying, updating, and locking object location databases, and overhead due to increased traffic, for instance for message forwarding.

The language microABS considered in Chapter 2 can be viewed as a candidate for a minimally functional object-based language. Essentially it allows to define a dynamically growing or shrinking collection of objects communicating by asynchronous RPC, and thus its functionality is not much different from a core version of Erlang [17], or the nomadic PICT language studied in [78]. The question is how program behaviour is affected by being run in the networked model, as compared with a more standard (reference) semantics given, for instance, using rewrite logic. This comparison is of interest, since the reference semantics is given at a high level of abstraction and ignores almost all aspects of physical distribution, such a location, routing, message passing, and so on. In Chapter 2 we show that, with a maximally nondeterministic networked semantics, and in the sense of barbed equivalence [73] which is a standard equivalence to study in these types of applications [19, 40, 43], programs exhibit the same behaviour in both cases.

Messaging in the microABS setting is very simple. The implicit channel abstraction used in the reference semantics for microABS in Chapter 2, and for the various ABS dialects in general, is essentially that of a reliable, unordered communication channel. Messages (calls) are transmitted according to the program order. Methods are received synchronously at any time, but in the fully asynchronous model considered here scheduling does not introduce any happens-before constraints on observable events arising from racing (i.e., simultaneously enabled) method calls. Soundness and full abstraction for the networked semantics is therefore an interesting and useful observation, since it allows many conclusions made at the level of abstract program behaviour to be transferred to the setting of a networked realization.

The question is how sensitive this observation is to the type of communication taking place at the abstract level. For this reason it is of interest to examine also languages with richer communication structures than asynchronous point to point message passing. To this end we enrich in this chapter the microABS language studied earlier with future variables and show that the conclusions of our previous work remain valid, however

---

1This is strictly speaking not true in general, as in the reference semantics, the program order on calls may induce happens-before constraints on external method calls that cannot be realized in the networked semantics, because messages are explicitly queued and can always be shuffled. However, barbed equivalence is not sensitive to this type of happens-before constraint in the reference semantics.
with more involved constructions.

Futures \[18, 39, 64, 87, 68, 29\] are placeholder variables for values that may be waiting to get instantiated. Futures are used extensively in concurrent and distributed high-level languages, libraries, and models including Java, .NET, Scheme, concurrent LISP, Oz, to name just a few. Our work relies on the future model introduced in \[29\] where futures are introduced as placeholders for return values of remote method calls, the model also underpinning the use of futures in ABS generally. Other models exist, such as the transparent futures considered in \[15\], or the concurrent constraint store model of Oz \[81\]. ABS has other constructs of relevance, such as the notion of concurrent object group (COG, \[76\]), but this is introduced in ABS mainly as a container for local evaluation, to support, within the context of a single computational unit, the collaborative scheduling model also introduced in \[29\]. Since our focus here is squarely on large scale message passing concurrency, it is therefore not too farfetched to claim that the language and model of asynchronous object networks with futures studied here is a reasonable candidate for a networked ABS core language.

The contribution reported in \[26\] and summarized in the present chapter, then, is an extension of the networked semantics of Chapter 2 to a richer fragment of ABS including futures, and a correctness proof showing that soundness and full abstraction remains valid in this richer setting.

The chapter is organized in the following way: In Section 3.2 we briefly introduce the extended version of microABS, called milliABS, and sketch its reduction semantics, as in Chapter 2. For details we refer the reader to the full version \[26\]. In Section 3.4 we present the network semantics, runtime states and the reduction relation, and in Section 3.5 we sketch the correctness (soundness and full abstraction) proof. We conclude the chapter by briefly discussing the implications of scheduling on the correctness analysis. Proofs are deferred to the full version \[26\].

### 3.2 mABS

We define a small concurrent object-based language milliABS with asynchronous calls and futures, as depicted in Figure 3.1. The milliABS language is an extension of the language microABS of message-passing processes introduced in the previous chapter with return values as futures. The addition of futures is the main difference in relation to microABS. The idea is that a method call, say, \texttt{foo!bar(x)} returns immediately with an uninitialized future \(f\), as a placeholder for a not-yet-defined return value, at the same time spawning an invocation of method \texttt{bar} on object \texttt{foo}, \texttt{foo.bar(x)}, that will at some later time cause the return value to become initialized. A task can at any time attempt to retrieve the value of a future \(f\), by executing \texttt{f.get}. If \(f\) is not yet instantiated this will cause the calling task to be suspended. The main benefit of futures is that long-running computations can be spawned off to execute in parallel without holding up the caller until the return value is actually needed. Futures are central to this work. One of our main objectives, above and beyond \[24\], is to show how futures can be implemented in a sound and fully abstract fashion on a large scale network in a fully decentralized and network transparent fashion.

**Example 3.2.1.** A very simple server applying \texttt{foo} to its argument is shown in Figure 3.2.
Example 3.2.2. Assume that \( \text{combine}(\text{upper}(x), \text{lower}(x)) = \text{foo}(x) \). The program example in Figure 3.3 returns immediately with the result, if the argument to \text{serve} is small. If the argument is not small, two new servers are spawned, and computation of the result on upper and lower tranches is delegated to those servers. The results are then fetched from the two newly spawned servers by evaluating the get statements, combined, and returned.

3.3 Reduction Semantics

We first present an abstract “reference” semantics for milliABS based on rewriting logic. The presentation follows Chapter 2 quite closely. We use the abstract semantics as the point of reference for the concrete network-oriented semantics which we present later. The goal is to show that the concrete network semantics correctly implements the abstract semantics in the sense of barbed equivalence. The reduction semantics uses a reduction relation \( cn \rightarrow cn' \) where \( cn, cn' \) are configurations, as determined by the runtime syntax in Figure 3.4. Later on, we introduce different configurations and transition relations, and so use index 1, or talk of e.g., configurations of “type 1”, for this first semantics when we need to disambiguate. We highlight only those aspects of the runtime syntax that are changed with respect to Chapter 2. The runtime syntax now has four types of containers: Tasks, objects, futures, and calls. Tasks and calls are essentially unchanged from the microABS semantics. Futures are used as centralized stores for assignments to future variables. Task and object environments \( l \) and \( a \), respectively, map local variables to values. Task environments are aware of a special variable \( \text{ret} \) that the task can use in order to identify its return future. Upon invocation, the task environment is initialized using the operation \( \text{locals}(o, f, m, v) \) by mapping the formal parameters of \( m \) in \( o \) to the corresponding actual parameters in \( v \), by initializing the method local variables to suitable null values, by mapping \( \text{self} \) to \( o \), and by mapping \( \text{ret} \) to \( f \), intended as the return future of the task being created. Object environments are initialized using the operation \( \text{init}(C, v) \), which maps the parameters of \( C \) to \( v \), and initializes the object local variables as above.

The reduction rules of Figure 2.5 are unchanged for milliABS. The remaining rules in 3.5, replacing those of Figure 2.6, address the more interesting cases that involve inter-object communication, external method
are evaluated, and get statements cause the evaluating task to hang until the value associated to the future

The second part of the chapter where we address the problem of efficiently executing milliABS programs on an abstract network graph using the name-based routing scheme introduced in Chapter 2. A commonplace approach to implementation of futures is by forward chaining. In ABS, futures are produced as placeholders for return values. Each object mentioning a future can subscribe to that future at some other object. This may happen in remote method calls where the caller subscribes to the return value later to be provided by the callee. It may also happen when a value containing a future is passed from some sender object to some receiver object. In that case the receiver object becomes subscriber at the sender object for that future. When a future gets instantiated to an actual value at some object, it is the task of that object to forward the instantiation to the subscribing objects. This is the implementation strategy applied in our work as well, and it is the objective of the proof to show that this approach is sound and fully abstract for our network semantics, even when routing is in an unstable state.

\[^{2}\]They are very meaningful in a labelled semantics setting, but that is a different story.
Other language models and implementation strategies exists, besides the one adopted here. It is, for instance, possible to lift futures to become first class objects with an explicit instantiation method. This design choice introduces the possibility of race conditions. In the ABS language this complication is avoided, as future instantiations are always rooted in some return statement, a property we rely on heavily in the technical development below. It is also possible to adopt a lazy strategy for instance propagation, instead of the eager strategy adopted here, where future instantiations are forwarded as soon as they become available, regardless of whether the subscriber of the instantiation actually has a need for the value. A possible alternative design is to instead request an instance when needed. This has the disadvantage of introducing additional delays at runtime, something which may nullify, to some extent, the rationale of introducing futures in the first place. This is discussed in more detail in the following chapter.

3.4.1 milliABS-NET Runtime Syntax

In Figure 3.6 we present the milliABS-NET runtime syntax, i.e., the shape of the runtime state. Recall from Section 3.3 that we reuse symbols as much as possible and use indices to disambiguate. Thus, for instance, Obj is the set Obj of the type 1 semantics in Figure 3.4, and Obj2 is the corresponding set in Figure 3.6. We adopt the same syntactical conventions as in Section 3.3. Tasks are unchanged from Figure 3.4. We write t(cn) for the multiset of tasks in cn, i.e., the multiset \{tsk \mid \exists cn'.cn = tsk cn'\}, and o(cn) for the multiset of objects in cn, similarly defined. We also write m(cn) for the multiset \{msg \mid msg \preceq cn\}.

We proceed to explain the different types of containers and the operations on them, highlighting the differences re. the treatment in Chapter 2. For a detailed explanation of other features such as routing, message passing, and the network graph, we refer to the previous chapter, and to [24] [26].

**Objects and Object Environments** Objects o(a, a, u, qin, qout) are now attached to a node u and a pair of an incoming (qin) and an outgoing (qout) fifo message queue, and the notion of object environment is refined to take futures into account in a localized manner. In the type 2 semantics, object environments a are now augmented by mapping futures fut to pairs (v⊥, o) where:

- v⊥ is the lifted value currently assigned to fut at the current object, and
- o is a forwarding set of the objects subscribing to updates to fut at the current object.

For instance, if a(fut) = (⊥, o₁ :: o₂ :: ε) the future fut is as yet uninstantiated (at the object to which a belongs), and, if fut eventually does become instantiated, the instantiation must be forwarded to o₁ and o₂, in random order.

We introduce some syntax to help manipulating object environments:

- a(x) abbreviates π₁(a)(x), a(f) abbreviates π₂(a)(f)
a[v/x] is a with \( \pi_1(a) \) replaced by the expected update. Similarly \( a[v/f] \) updates \( \pi_2(a) \) by mapping \( f \) to the pair \( (v, \pi_2(a(f))) \), i.e., the assigned value is updated and the forwarding list remains unchanged. If \( f \notin dom(\pi_2(a)) \) then \( a[v/f](f) = (v, \varepsilon) \), i.e., the update to value takes effect. Finally we use \( a[(v, o)/f] \) for the expected update where both the value and the forwarding list is updated.

\( fu(v, o, a) \) updates \( \pi_2(a) \) by for each future \( f \) occurring in \( v \) adding \( o \) to the forwarding list of \( a(f) \), i.e., by mapping \( f \) to the pair either \((\bot, o)\) if \( a(f) \) is undefined \((= \bot)\), or \((\pi_1(a(f)), o :: \pi_2(a(f)))\) otherwise.

\( init(C, v) \) returns an initial object environment by mapping the formal parameters of \( C \) to \( v \).

\( init(f, a) \) augments \( a \) by mapping \( f \) to the pair \((\bot, \varepsilon)\). If \( f \notin dom(a) \) then \( init(f, a) = a \).

\( init(v, a) \) augments \( a \) by mapping each \( f \) in \( v \) which is uninitialized in \( a \) (i.e., such that \( f \notin dom(a) \)) to \((\bot, \varepsilon)\).

As a consequence of this change, futures are eliminated as containers in the type 2 runtime syntax. In other respects, the type 2 runtime syntax is unchanged: Syntactical conventions that are not explicitly modified in the type 2 syntax above are unchanged, in particular we continue to assume multiset properties of configuration juxtaposition.

**Messages** In relation to Chapter 2 call messages are modified slightly, to the shape \( \text{CALL}(o, o', f, m, v) \) in order to record the identity of the associated future, and a future instantiation message \( \text{FUTURE}(o, f, v) \) with destination \( o \) is added. Both call messages and future messages are object bound. Other concepts such as object closures are unchanged from Chapter 2.

### 3.4.2 Reduction Semantics

As noted in Chapter 2, an important distinction between the standard semantics and the network semantics is the absence of binding. For the standard semantics, name binding plays an important role to avoid clashes between locally generated names. However, in a language with NIDs this device is no longer needed, as globally unique names can be guaranteed easily by augmenting names with their generating NID. Since all name generation in the milliABS-NET semantics below takes place in the context of a given NID, we can simply assume operations \( \text{newf}(u) \), resp., \( \text{newo}(u) \), that return a new future, resp., OID, which is globally fresh for the “current context”. We use \( \text{newf}(z) \) for either \( \text{newf} \) or \( \text{newo} \) when the nature of \( z \) is not known.

We present the milliABS-NET reduction rules. First, figure 3.5 applies with the following two minor modifications:

- Rule CTXT-2 is dropped as name binding is dropped from the type 2 runtime syntax.
- Rule WFIELD is modified in the obvious way to read: If \( x \in \text{dom}(a) \) then \( o(a, a, u, q_{in}, q_{out}) \ t(o, l, x = e; s) \rightarrow o(a, a[e(a, l)/x], u, q_{in}, q_{out}) \ t(o, l, s) \)

Next, the rules in Figure 3.6 also appear unchanged, with the exception of rules CALL-SEND and CALL-RCV. These are replaced by the six rules in Figure 3.7. The four rules CALL-SEND, CALL-RCV, FUT-SEND, FUT-RCV produce and consume messages, method calls and future instantiations. A method call causes a local future to be created and passed with the call message. Upon reception of the call, the callee first initializes those received futures it does not already know about, and then augments the resulting local object environment to forward instantiations of the received future to the caller. Observe that it may be that the callee already knows about the return future of the call. Since message order is not assumed to be preserved a later call referring to the original return future may overtake the earlier call. The eventual return value becomes bound to the return future by the assignment to the constant \( \text{ret} \) during initialization of the called methods local environment. The rule FUT-SEND may cause future instantiations to be forwarded to objects in the forwarding list whenever the future is seen to have received a value, and FUT-RCV causes the receiving object
CALL-SEND: Let $o' = e_1(a, l)$, $v = e_2(a, l)$, $f = \text{newf}(u)$ in
$$o(a, u, q_{in}, q_{out}) \xrightarrow{t(o, l, x = e_1!m(e_2); s)} o(a, fu(v, o', init(f, a)), u, q_{in}, enq(\text{CALL}(o, o', f, m, v), q_{out})) \xrightarrow{t(o, l/f/x), s}$$

CALL-RCV: If $hd(q_{in}) = \text{CALL}(o', o, f, m, v)$ then $o(o, a, u, q_{in}, q_{out}) \rightarrow o(o, a, u, q_{in}, q_{out})$
$$o(a, fu(f, o', init(v, init(f, a))), u, deq(q_{in}), q_{out}) \xrightarrow{t(o, locals(o, m, f, v), body(a, m))}$$

FUT-SEND: If $a(f) = (v, o_1 :: o_2)$ then
$$o(a, u, q_{in}, q_{out}) \rightarrow o(a, fu(v, o', a[(v, o_2)/f]), u, q_{in}, enq(FUTURE(o_1, f, v)), q_{out})$$

FUT-RCV: If $hd(q_{in}) = FUTURE(o, f, v)$ then
$$o(a, u, q_{in}, q_{out}) \rightarrow o(a, a[v/f], u, deq(q_{in}), q_{out})$$

RET-2: $o(a, a, u, q_{in}, q_{out}) \xrightarrow{t(o, l, \text{return } e; s)} o(a, a[\hat{e}(a, l)/l(\text{ret})], u, q_{in}, q_{out})$

GET-2: If $\hat{e}(a, l) = f$ and $a(f) = v$ then
$$o(a, a, u, q_{in}, q_{out}) \xrightarrow{t(o, l, x = e.get; s)} o(a, l[v/x], s)$$

Figure 3.7: milliABS-NET reduction rules

to update its local environment accordingly. A future may itself be instantiated to a future. The local forwarding table may thus need to be updated.

The two rules RET-2 and GET-2 handle the corresponding language constructs. Return statements cause the corresponding future to be instantiated, as explained, and get statements read the value of the future provided it has received a value.

We leave again the definitions of initial configuration, reachability, and type 2 well-formedness to [26]. It must be noted that the well-formedness conditions become rather more subtle in the type 2 case, as various consistency properties of the network semantics must be verified, for instance to ensure that assignments to futures are always consistent. Whereas this is trivial in the type 1 case since futures there are uniquely represented, in the type 2 case a much more involved analysis is needed.

### 3.5 Normal Forms

An milliABS-NET program can be run from an initial state in either the type 1 or the type 2 semantics. We want to show that the behaviour of the programs is preserved, in the sense that the initial states at type 1 and type 2 are barbed equivalent, referring to Chapter 2 for the definition of barbed equivalence.

The key to the proof is a normal form lemma for milliABS-NET saying, roughly, that any well-formed type 2 configuration can be rewritten into a form where queues have been emptied of all routable messages, where routing tables have been in some expected sense normalized, where all futures that are assigned a value somewhere are assigned a value everywhere the value might be needed (by well-formedness this value is unique), and where all objects have been moved to a single node. We perform this rewriting in two steps:

- First we show that routing can be stabilized and inter node links emptied, except for external messages (messages addressed at an external OID). This part is unchanged from Chapter 2 and not repeated here.

- We then complete the construction by emptying object queues, propagating futures, and moving all objects to a single node. This is done in the following section.
Algorithm 3: Normalization

Input: Well-formed type 2 configuration \( cn \) on a connected network graph

Output: Configuration in type 2 normal form

fix a NID \( u \);
run Algorithm 1;
while some object queue is nonempty {
    use \textsc{msg-send}, \textsc{msg-delay-2}, \textsc{call-rcv}, \textsc{fut-rcv} to dequeue one
    message from each nonempty object queue;
    while \textsc{fut-send} is enabled { apply \textsc{fut-send} } ;
    while an object exists not located at \( u \) {
        use \textsc{obj-send} to send the object towards \( u \);
        run Algorithm 1}
}

Figure 3.8: Algorithm 3 – Normalization

3.5.1 Normalization

The normalization procedure, Algorithm 3, shown in Figure 3.8 is a minor extension of alg. 2 of the previous section. Let \( A_3(cn) \rightarrow cn' \) if \( cn' \) is a possible result of applying Algorithm 3 to \( cn \). Initially a node \( u \) is chosen towards which all objects will migrate during normalization. Normalization is performed in cycles, each cycle starting and ending in a stable configuration. In each cycle one message is read from the object in- and out-queues. By well-formedness, object queues contain only calls and future messages. Receptions of future messages may cause the object environment to instantiate futures. This may cause new future instantiation messages to be enabled. Accordingly, those messages are generated and sent to the objects out-queue. Once this is done, objects not yet at \( u \) will be migrated.

Proposition 3.5.1. Algorithm 3 terminates \( \square \)

We then turn to normal forms and define first a couple of auxiliary operations. Let \( t_2(cn) \) be the multiset of method containers \( tsk = (o,t,s) \) such that one of the following cases apply:

- \( tsk \) is a task container in \( cn \).
- There is a message \( \text{call}(o',o,f,m,v) \) in transit, \( o \) is not external, \( l = \text{locals}(o,m,f,v) \) and \( s = \text{body}(o,m) \).

Let \( o_2(cn) \) be the multiset of object containers \( o(o,a,u,\epsilon,\epsilon) \) for which the following apply:

- There is an object container \( \text{obj} = o(o,a',u',q_{\text{in}},q_{\text{out}}) \leq cn \)
- \( a'(x) = a(x) \) for all variables \( x \)
- \( a'(f) = (v,\epsilon) \) with \( v \neq \perp \) if, and only if, for some object container \( o(a_1,a_1,u_1,q_{\text{in},1},q_{\text{out},1}) \leq cn, \) \( a_1(f) = (v,o) \)

Also say that \( cn \) has \textit{external messages only}, if link queues in \( cn \) contain only external messages.

Definition 3.5.2 (Normal Form). A well-formed configuration \( cn \) is in normal form, if

1. \( cn \) has stable routing
2. \( cn \) has external messages only
3. \( t(cn) = t_2(cn) \)
4. \( o(cn) = o_2(cn) \)
Proposition 3.5.3. Suppose \( cn \) is well-formed. If \( A_3(cn) \leadsto cn' \) then

1. \( cn' \) is in normal form
2. \( \text{graph}(cn) = \text{graph}(cn') \)
3. \( t_2(cn) = t(cn') \)
4. \( o_2(cn) = o(cn') \)
5. \( m_1(cn) = m(cn') \)

Proposition 3.5.3 motivates the following definition of normal form equivalence.

Definition 3.5.4 \( (\equiv_2) \).

1. Let \( cn_1 \equiv_2 cn_2 \) if and only if \( cn_1 \) and \( cn_2 \) are both well-formed, \( \text{graph}(cn_1) = \text{graph}(cn_2) \), \( t_2(cn_1) = t_2(cn_2) \), \( o_2(cn_1) = o_2(cn_2) \), and \( m_1(cn_1) = m_1(cn_2) \).

2. Let \( cn_1 \equiv_2 cn_2 \) if and only if there are \( cn'_1, cn'_2 \) such that

\[
A_3(cn_1) \leadsto cn'_1 \quad R_2 \quad cn'_2 \leadsto A_3(cn_2)
\]

We obtain that normalization respects normal form equivalence.

Corollary 3.5.5. If \( A_3(cn) \leadsto cn' \) then \( cn \equiv_2 cn' \)

The following lemma is central to the use of normal forms.

Lemma 3.5.6. \( \equiv_2 \) is reduction closed.

Proposition 3.5.7. If \( cn_1 \equiv_2 cn_2 \) then \( cn_1 \simeq cn_2 \).

Corollary 3.5.8. If \( A_3(cn) \leadsto cn' \) then \( cn \simeq cn' \).

3.6 Correctness

In this section we prove correctness of the network semantics by mapping each well-formed type 1 configuration \( \text{bind } z.cn \) in standard form to a well-formed type 2 configuration \( \text{down}(cn) \) on an arbitrary underlying network graph. We then prove that the two configurations are barbed equivalent, i.e., that \( \text{bind } z.cn \simeq \text{down}(cn) \).

Defining the Underlying Network Graph  We first fix an underlying graph represented as a well-formed type 2 configuration \( cn_{\text{graph}} \) with a distinguished UID \( u_0 \), similar to the way initial configurations are defined in Section 3.4. Thus, \( cn_{\text{graph}} \) consists of nodes and links only, each node \( u \) in \( cn_{\text{graph}} \) has the form \( (u,t) \), and each link has the form \( (u,\varepsilon,u') \). The routing tables \( t \) are defined later.
Representing Names and Values  To represent names, one complication is that names in the type 1 semantics need to be related to names in the type 2 semantics, which does not include the binding construct of the type 1 semantics, but on the other hand has different generator functions (the functions newf and newo). This prevents the name relation from being modeled using the identity relation. To address this we assume that names and futures in the type 1 semantics are really symbolic, connected to concrete names/futures used in the type 2 semantics by means of an injective name representation map rep, taking internal names f, o in the type 1 semantics to names rep(f), rep(o) in the type 2 semantics. For convenience we extend the name representation map rep to arbitrary values and task environments as follows:

- \( rep(o) = o, \) if \( o \in \text{Ext} \),
- \( rep(p) = p, \) if \( p \in PVal \),
- \( rep(l)(x) = rep(l(x)), \) \( rep(l)(\text{ret}) = rep(l(\text{ret})) \)

Representing Object Environments  To extend rep also to object environments, a complication is that object environments in the type 2 semantics must be defined partially in terms of the type 1 environments (for object variables) and partially in terms of the future containers available in the “root configuration”, since the type 1 semantics uses future containers in place of forwarding lists stored in object environments. To this end we first define an auxiliary function oenvmap\((cn, p, rep) \in \text{Fut} \to \text{Val}_{\bot} \) on triples of type 1 configurations, a pool of OID/future constants, and a name representation map, as a function which gathers together assignments to futures as determined by the future containers in \( cn \):

- \( \text{oenvmap}(0, p, \text{rep})(f) = \text{oenvmap}(\text{tsk}, p, \text{rep}) = \text{oenvmap}(\text{obj}, p, \text{rep}) = \text{oenvmap}(\text{cl}, p, \text{rep}) = \bot \)
- \( \text{oenvmap}(f(f, v_{\bot}), p, \text{rep})(f') = (\text{if} \ \text{rep}(f) = f' \ \text{then} \ \text{rep}(v_{\bot}) \ \text{else} \ \bot) \)
- \( \text{oenvmap}(\text{bind } o.\text{cn}, p \cup \{o'\}, \text{rep})(f) = \text{oenvmap}(\text{cn}, p, \text{rep}[o'/o])(f) \)
- \( \text{oenvmap}(\text{bind } f.\text{cn}, p \cup \{f'\}, \text{rep})(f'') = \text{oenvmap}(\text{cn}, p, \text{rep}[f'/f])(f'') \)
- \( \text{oenvmap}(\text{cn}_1 \ \text{cn}_2, p, \text{rep})(f) = \text{oenvmap}(\text{cn}_1, p, \text{rep})(f) \cup \text{oenvmap}(\text{cn}_2, p, \text{rep})(f) \)

Fix now a root type 1 configuration \( cn_0 \) and a large enough pool \( p_0 \) of names (proportional to the size of \( cn_0 \), and computed using newf\((u_0)\) and newo\((u_0)\) to conform with our naming policy). Assume that \( cn_0 \) is in standard form, i.e., \( cn_0 = \text{bind } z_0.\text{cn}'_0 \) where \( \text{cn}'_0 \) does not have binders. Fix \( g = \text{oenvmap}(\text{cn}_0, p_0, \bot) \) and \( \text{cn}_{\text{graph}} \) as above. We can now extend rep to object environments by:

- \( \pi_1(\text{rep}(a))(x) = \text{rep}(\pi_1(a))(x) \)
- \( \pi_2(\text{rep}(a))(f) = \left\{ \begin{array}{ll} (g(f), \varepsilon) & \text{if } g(f) \neq \bot \\ (\bot, \text{OID}(cn_0)) & \text{otherwise} \end{array} \right. \)

Representing Call Containers  Another complication is that we need an operation to represent a type 1 call container as a message in the type 2 semantics. This is done in the obvious way by the operation send as follows:

\[
\begin{align*}
\text{send} & (\text{c}(o, o', f, m, \to), \text{of}(a, a, u, q_{in}, q_{out}) \ \text{cn}) \\
& = \ \text{of}(a, a, u, q_{in}, \text{enq}(\text{CALL}(o, o', f, m, \to), q_{out})) \ \text{cn}
\end{align*}
\]
Representing Configurations  Given a name representation map rep we can now define the representation of a type 1 configuration as a transformer on type 2 configurations, initially the underlying network graph, as the mapping down as follows:

- \( \text{down}(0, \text{rep})(cn) = cn \)
- \( \text{down}(cn_1, cn_2, \text{rep}) = \text{down}(cn_1, \text{rep}) \circ \text{down}(cn_2, \text{rep}) \)
- \( \text{down}(t(o, l, s), \text{rep})(cn) = t(\text{rep}(o), \text{rep}(l), s) \ cn \)
- \( \text{down}(o(o, a), \text{rep})(cn) = o(\text{rep}(o), \text{rep}(a), u_0, \varepsilon, \varepsilon) \ cn \)
- \( \text{down}(f(f, \rightarrow \bot), \text{rep})(cn) = cn \)
- \( \text{down}(c(o, o', f, m, \rightarrow), \text{rep})(cn) = \text{send}(c(o, o', f, m, \rightarrow), cn, u_0) \)

In other words, we represent type 1 configurations by first assuming some underlying network graph, secondly distributing the (centralized) assignments to futures in each object environment with the trivial forwarding lists, and then, once this is done, mapping the containers individually to type 2 level.

Defining Routing Tables  The only detail remaining to be fixed above is the routing tables. For \( u_0 \) the initial routing table, \( t_0 \), needs to register all objects in \( cn_0 \), i.e.,

\[
t_0 = \text{reg}(g(o_0), u_0, \text{reg}(g(o_1), u_0, \text{reg}(\ldots, \text{reg}(g(o_m), u_0, \bot))\ldots))
\]

where \( o_0, \ldots, o_m \) are the OIDs in \( cn_0' \). For nodes \( n(u, t) \) where \( u \neq u_0 \) we let \( t \) be determined by some stable routing. This is easily computed using Algorithm 1, and we leave out the details.

Definition 3.6.1 (Representation Map down). Let a network graph \( cn_{\text{graph}} \) and a name representation map rep be given. For each well-formed type 1 configuration \( cn_0 \), the type 2 representation of \( cn_0 \) is the configuration \( \text{down}(cn_0) = \text{down}(cn_0, \text{rep})(cn_{\text{graph}}) \).

In this definition, forwarding lists are overapproximated as compared to the type 2 semantics, which forward futures only to objects that have actually received them. To handle this slight complication we need a little lemma saying that for well-formed type 2 configurations, forwarding lists can be extended without affecting observable behaviour. To make this precise say that \( o(o, a, u, q_{in}, q_{out}) \) extends \( o(o', a', u', q'_{in}, q'_{out}) \), if \( o = o' \), \( u = u' \), \( q_{in} = q'_{in} \), \( q_{out} = q'_{out} \), \( a(x) = a'(x) \) for all \( x \), and \( \pi_1(a(f)) = \pi_1(a'(f)) \) and \( \pi_2(a(f)) \supseteq \pi_2(a'(f)) \) for all \( f \).

Lemma 3.6.2. Suppose that \( cn \) is type 2 well-formed, and \( cn' \) differs from \( cn \) only by replacing each object \( \text{obj} \) by an object \( \text{obj}' \) such that \( \text{obj}' \) extends \( \text{obj} \). Then \( cn' \) is type 2 well-formed as well, and \( cn \simeq cn' \). \( \square \)

As in Chapter 2 we can now show a key lemma allowing us to relate transitions in the two semantics under barbed equivalence.

Lemma 3.6.3. Let \( \text{bind } z.cn \) be type 1 well-formed in standard form.

1. If \( \text{bind } z.cn \rightarrow \text{bind } z'.cn' \) then \( \text{down}(cn) \rightarrow^* \circ \simeq \text{down}(cn') \)
2. If \( \text{down}(cn) \rightarrow cn'' \) then for some \( z', cn', \text{bind } z.cn \rightarrow^* \text{bind } z'.cn' \) and \( cn'' \simeq \text{down}(cn') \) \( \square \)

We can now state the main result.

Theorem 3.6.4 (Correctness of the Type 2 Semantics). For all well-formed type 1 configurations \( cn \) on a connected network graph,

\[
 cn \simeq \text{down}(cn) \]

\( \square \)
3.7 Scheduling

The type 2 semantics is highly nondeterministic. The semantics says nothing about how frequently routing tables are to be exchanged, when messages should be passed between the different queues, when future messages are to be sent, and when, and to where, objects are to be transmitted. Resolving these choices is a crucial tradeoff between management overhead and performance. For instance, if routing tables are exchanged at a very high frequency, routing can be always assumed to be in stable state. This ensures short end to end routes, but at the expense of a large management overhead. This raises the question of how to determine these parameters, something which we address in more detail in the following chapter.

Regardless how this is done, a real implementation needs to resolve these choices. This is tantamount to eliminating nondeterminism from the type 2 semantics, essentially by removing transitions. Thus, in a sense, Theorem 3.6.4 achieves more than is called for, as soundness and full abstraction a priori applies only to the type 2 semantics with all transitions included.

A scheduler can be viewed abstractly as a predicate on histories in the following way. Let a scheduled execution be any sequence \( \epsilon = c_0c_1\cdots c_n \) such that \( c_i \rightarrow c_{i+1} \) for all \( 0 \leq i < n \) where the \( c_n \) are well-formed type 2 configurations. A scheduler is a predicate \( S \) on such sequences, with the property that

1. \( S(\langle c \rangle) \) for any \( c \) where \( \langle c \rangle \) is the one element execution consisting of \( c \) (a scheduler kicks in only when execution is started), and

2. if \( S(c_0\cdots c_n) \) and \( c_n \rightarrow c_{n+1} \) for some \( c_{n+1} \) then \( S(c_0\cdots c_n c_{n+1}) \) for exactly one \( c_{n+1} \).

Then a scheduled type 2 semantics is a transition system on executions \( \epsilon = c_0\cdots c_n \) such that \( \epsilon \rightarrow \epsilon' \) if and only if \( \epsilon' = c_0\cdots c_n c_{n+1} \) and \( S(\epsilon') \).

Define now the barbed simulation preorder \( \sqsubseteq \) on executions by requiring the existence of a witness relation \( R \) which satisfies reduction closure and barb preservation (when \( c_0\cdots c_n \downarrow \text{obs} \) if and only if \( c_n \downarrow \text{obs} \)), but not necessarily symmetry. We immediately obtain from Theorem 3.6.4 the following corollary:

Corollary 3.7.1. For all well-formed type 1 configurations \( c \) on a connected network graph,

\[
\down{(\text{down}(c))} \sqsubseteq c.
\]

Proof. It suffices to note that if \( \epsilon = c_0\cdots c_n \rightarrow \epsilon' = c_0\cdots c_{n+1} \) then \( c_n \rightarrow c_{n+1} \), and if \( \epsilon \downarrow \text{obs} \) then \( c_n \downarrow \text{obs} \) as well. \( \square \)
Chapter 4

ABS-NET: Fully Decentralized Runtime Adaptation for Distributed Objects

4.1 Introduction

One motivation for decoupling computational processes and physical infrastructure in a distributed setting is that it becomes possible to handle resource allocation at layers lower than the application layer. Potentially, tasks can then be performed at the physical machine most suited at the moment, continually meeting global system requirements for e.g. even utilization and task-local requirements such as a limited response time.

We consider the problem of runtime adaptation of tasks in the context of the Core ABS fragment of ABS \[56\]. In the semantics of Core ABS, no account is made of location or relation to a physical infrastructure. In order to account for issues related to location, object migration and physical infrastructure constraints, we extend Core ABS runtime configurations to include an abstract graph representation of a network, and extend runtime behaviour to include mobility of objects between network nodes, similarly to in previous chapters. Conceptually, adaptation is made possible by a controller process running on each network node, which communicates with controllers at other nodes but makes independent decisions on object mobility based only on local data.

To enable precise reasoning and experiments on adaptivity, we define three central Quality of Services (QoS) objectives which a solution for runtime adaptation in our context can be assessed against: node load, arc load and message latency. We abstract from many implementation-level concerns when interpreting these objectives in our setting. The load for a specific node at a specific time is simply the number of active tasks running on it. The load for a specific arc is the number of messages traversing the arc. The latency for a specific message is the number of hops needed to reach its destination. We then restrict our consideration of adaptivity to the problem of how to ship objects around to achieve the objectives as well as possible, given a specific static network topology, ABS program, and node-local procedure for managing migrations.

Using a simulator which implements the key parts of our semantics, we have investigated how well objectives are fulfilled for some application-relevant choices of network topologies, programs and migration procedures. In future work, we plan to extend the investigation to dynamic networks.

4.2 Core ABS Adaptation

The runtime unit of concurrency in Core ABS is a concurrent object group (cog). A cog contains one or more runtime objects, which perform cooperative scheduling of tasks. We use a variant of Core ABS where a single object is the unit of concurrency rather than a cog, similarly to Albert et al. \[3\]. The choice is motivated by our focus on network adaptability of individual objects and computation tasks, which becomes more complicated when objects in a group must perform intermittent synchronization. In this language variant, all individual objects can be interpreted as actors, having local store and communicating with the environment only via asynchronous message passing. Additionally, our language variant fixes a number of
minor inconsistencies in the syntax and semantics of the original Core ABS, for example by prohibiting multiple return statements which could cause unexpected nonterminating behaviour. The language variant is described in detail in the previous chapter, see also [25].

A fragment of a Core ABS program is given as an example in Figure 4.1. The CastNode interface defines a method aggregate, which, when called on some object, is intended to perform a convergecast operation in the binary tree rooted at that object. Specifically, this means that if an object implementing CastNode is a leaf in the tree (an instance of class LeafNode), it simply returns a locally known integer, but if the object has child nodes in the tree (an instance of class BranchNode), aggregate is called on both of those objects and the results are added to the local integer and returned. In this way, the aggregate method for the object $o$ always returns the aggregate of all local values in the binary tree of objects rooted at $o$.

The implementation of the aggregate method in the program highlights the use in Core ABS of futures as placeholders for results from asynchronous method calls. The variables $f_{Left}$ and $f_{Right}$ hold futures which ultimately resolve to integer values, as indicated by their type declarations. In the right hand side of the declarations of the futures, the delimiter ‘!’ between the object variable name and the method name signifies asynchronous invocation, which always immediately returns a future. The usual dot delimiter ‘.’ signifies a synchronous invocation which blocks the caller until the final result is returned without any intermediary.

Before returning the aggregate of the current object, the aggregate of each child node is retrieved by appending .get to the variable holding the respective future. Evaluations of assignments with this construct can be blocking, unless an await statement was executed first with the future variable involved, e.g. await $f_{Left}$?. Executing await when the associated future has not yet been resolved does not force the caller into busy waiting; if there are method invocations for the object waiting be processed, control can be changed to the corresponding process at the discretion of the scheduler, and pass back to the original invocation later.

Informally, a Core ABS runtime configuration is a bag of objects and futures equipped with unique identifiers, along with unprocessed method invocations. An object in the bag has values for all variables defined in its class, a queue of processes representing received method invocations, and possibly an active process. Futures either have the value to which they resolve, or a placeholder to indicate that no resolution is available. When an asynchronous method invocation statement is executed, a method invocation is added to the bag, ready to be consumed by the callee. In contrast with actor languages such as Erlang and Rebeca [80], which provide the traditional guarantee that messages from one actor to another are always processed in the order they are sent, the Core ABS semantics does not prescribe any particular order for processing method invocations. In effect, the runtime environment provides an unbounded number of one-place buffers that objects can use to communicate with objects for which identifiers are known.

While interface names are proper type names in Core ABS, class names are not, and are thus only used in object creation with the new keyword. For example, the assignment $\text{CastNode} \ nd = \text{new} \ \text{LeafCastNode}(0)$; creates a LeafCastNode object with the val variable set to 0.

### 4.3 Network Model and Semantics

To reason about object adaptability to environmental conditions, we bring selected parts of the infrastructure of a distributed system into our model, namely, network endpoints and links. Endpoints and links are modelled as graph nodes and arcs with message queues, respectively. Conceptually, we consider a node to consist of an object layer, where local objects reside, and a node controller, which acts as a mediator between the environment and node-local objects, as illustrated in Figure 4.2. This node controller is not treated explicitly in previous chapters. The dashed arrow in the figure signifies that an object identifier is known by another object and thus can be used for method invocation. The node controller also contains logic for decision-making on adaptivity. Seen abstractly, adaptivity here becomes the problem of shipping objects around in the network graph to achieve an allocation that achieves a Quality of Service objective—with the added constraint that all reallocations must be decided locally at each node.
interface CastNode {
    Int aggregate();
}

class LeafCastNode(Int val) implements CastNode {
    Int aggregate() {
        return val;
    }
}

class BranchCastNode(Int val, CastNode left, CastNode right) implements CastNode {
    Int aggregate() {
        Fut<Int> fLeft = left!aggregate();
        Fut<Int> fRight = right!aggregate();

        Int aggregateLeft = fLeft.get;
        Int aggregateRight = fRight.get;

        return val + aggregateLeft + aggregateRight;
    }
}

Figure 4.1: Core ABS example interface and classes

Figure 4.2: Nodes, node controllers, and interpreter layers

4.3.1 Operational Semantics

We have defined a structured operational semantics \cite{25}, building upon the functional and object levels of our variant of Core ABS, which describes the dynamics of executing an ABS model on nodes in a network at a high level of abstraction. The semantics extends the semantics of Dam \cite{24, 26} to cover a larger fragment of Core ABS. Adaptability features such as routing information dissemination and object mobility are modelled as nondeterministic events, with the node controller consisting of nothing more than a globally unique identifier and a routing table. We refer to the combination of the Core ABS functional layer, Core ABS object syntax, the network-oriented runtime configuration syntax described below, and the associated structural operational semantics described below as ABS-NET. We intend for the semantics to both guide implementation, by defining a baseline for retaining program runtime behavioural similar to Core ABS in a distributed setting, and provide opportunities for further theoretical analysis of specific adaptability strategies by refinement.

39
4.3.2 Assumptions for Adaptability

In the semantics, the Core ABS program being executed is assumed to be available unaltered at all nodes. The program is therefore not explicitly represented in a runtime configuration. We consider only networks that remain static over the course of program execution. Handling benignly dynamic networks is a planned extension, but we leave all details of crash failures and byzantine failures for future work. On the same note, we also assume that messages sent between neighbour nodes cannot be lost—only ignored indefinitely as far as fairness permits.

We also assume that the behaviour of the program running on network nodes is nonterminating and cyclical. This assumption is motivated by our focus on adaptability; for adaptations to current conditions to have a chance of conveying benefits, similar conditions must hold in the future. Equivalently, if future conditions are random independently of current conditions, there is no obvious payoff in an adaptation strategy.

4.3.3 Node Controller Behaviour

The node controller’s relationship with the interpreter layer residing on the node is symbiotic. On one hand, the node controller provides message delivery services and callback functions to obtain new globally unique object identifiers for objects residing in the interpreter layer. On the other hand, the node controller triggers object movement by using callback functions that the interpreter layer makes available. We assume the node controller is aware, through its interaction with underlying network layers, of all nodes adjacent to the node it resides on, and can communicate with node controllers at neighbouring nodes. As mentioned earlier, in the model, such communication takes place through a buffer at an arc. In addition, each node controller is equipped with a self-loop arc that serves as the default route for messages that cannot immediately be routed to an adjacent node. Since there is no upper bound enforced on communication delays, the node controller always runs the risk that information received from the outside world is out of date.

The baseline behaviour for a node controller is defined by a reduction relation in the same style as Core ABS. Exchange of data with the interpreter layer, containing the local objects, is made explicit via the use of parameterized labels. For example, mobility involves an object being transported from the interpreter layer into a message passing from a node to an adjacent node.

4.3.4 Object Behaviour

Compared to Core ABS, objects at runtime are extended with input and output queues as in microABS and milliABS, as well as a map for storing resolved future values. The reduction rules in the Core ABS semantics which involve only a single object and its internal state have been transferred essentially unchanged into unlabelled interpreter layer reduction rules. In contrast, transitions that involve multiple objects or futures are translated into either transitions involving message passing or labelled rules for exchanging data with the node controller.

4.4 Adaptation

We consider three QoS objectives which runtime adaptation solutions can be assessed against: node load, arc load and message latency.

In our setting, the definition of node load is simple but coarse grained: the load on a node \( u \) is the number of objects located on \( u \) with active tasks. One advantage of this measure is that it is an intrinsic property of runtime configurations, rather than something extrinsic to our model such as processor load or the loadavg measure available in many Unix operating system variants. We need a model-intrinsic measure of load to enable reasoning at an abstract level about convergence to balanced allocations and that loads stay within a certain range. One disadvantage of the approach is that it fails to take into account the varying use of memory and processing power among tasks. However, in an implementation, a more fine-grained measure of load can be adopted, as long as it is linear in the number of active tasks.
We define the load of a particular arc as the number of messages traversing it. Hence, global minimization of arc load means that a minimal number of inter-node messages are sent overall, with respect to the current state of routing tables at nodes. Unless all routing tables are optimal (minimum stretch), however, there is no guarantee that the number of hops, i.e., latency, of a particular object-addressed message is minimal.

4.4.1 Node Load Balancing

Although we wish to simultaneously meet all of our QoS objectives fully, we consider node load balancing our primary concern. Load balancing solutions are also relatively well-studied in the literature, making it easier to find a good starting point.

Azar et al. [8] consider the problem of achieving balanced allocations in the framework of stochastic processes, where it is viewed as stepwise allocations of balls into bins. They highlight the use of greedy schemes for quickly converging to a ball-to-bin assignment where the maximum number of balls in any bin is minimized. The main drawback of this approach in a distributed setting is the reliance on atomic, single assignments of a ball to a bin at each algorithm step. Even-Dar and Mansour [36] study load balancing in a distributed setting where allocations are not necessarily done one-at-a-time. They give a distributed algorithm for selfish rerouting that quickly converges to a Nash equilibrium, which corresponds to a balanced resource allocation. However, at each round, locally computing a new allocation requires knowing precisely all loads in the system, which is complicated and costly to find out in the current setting.

Berenbrink et al. [11] describe and analyze fully distributed algorithms which require only local knowledge of the total number of resources and the load of one other resource to perform a single task migration step. The algorithms, some of which have attractive expected time for convergence, can be straightforwardly translated to a synchronous, round-based distributed setting, and further, e.g., via synchronizers [6], to a fully asynchronous setting. One important assumption made in the algorithm analysis is that a task can migrate to any other resource in a single concurrent round. For this property to hold, the underlying network graph must be complete, which we do not generally assume.

A factor in the convergence time is whether neutral moves are allowed, i.e., whether a migration can happen even when, as far as can be told locally, the move does not result in a more balanced allocation but merely an equally good one. If the network graph is sparse, and the number of active tasks an order magnitude greater than the number of nodes, allocations where the difference in load between any two neighbours is one but the maximal load difference is in the order of the graph diameter are possible. Such allocations clearly cannot be improved upon without neutral moves.

The problem of oscillating behaviour during task balancing can be mitigated by the use of coin flips before finalizing decisions to migrate tasks, as in the algorithms of Berenbrink et al. Oscillation can be made worse by information becoming stale, which is a fact of life in fully asynchronous systems. If the information is not too stale, however, the number of oscillation periods can sometimes be bounded [38].

4.4.2 Minimizing Communication and Other Objectives

The literature on load balancing related to scientific computing contains work on simultaneously optimizing task allocations and communication overhead. For example, Cosenza et al. [23] give a distributed load balancing scheme for simulations involving agents moving in space from worker to worker. The scheme, which is validated experimentally, optimizes both worker load and communication overhead between workers, but assumes only a small area of interest for each agent, with agents unable to communicate with other agents outside this area. In the current work, objects can communicate whenever object identifiers are known to the sender, making it harder to minimize communication overhead. Catalyurek et al. [20] describe how to use hypergraph partitioning to minimize both communication volume and migration time of tasks for parallel scientific computations. However, the repartitioning is performed in batch and requires complete, immediate knowledge of the data and computations on each node.

Querying the load of neighbours before deciding where to migrate an object can be costly in terms of arc load, and information received previously may not be accurate. Many load balancing algorithms therefore
have as a feature that the number of load queries sent is minimal when migrating a resource. A third measure which is discussed in the literature which we do not consider is the cost in terms of time and messaging for migration itself.

4.5 Evaluation

We have evaluated ABS-NET by developing a simulator for running ABS programs in a network of nodes according to our semantics. We have run the simulator with a variety of network node topologies, programs and object migration policies.

4.5.1 Simulator

Our simulator’s main purposes are to serve as a proof-of-concept for ABS-NET, and to allow us to run various adaptability case studies with particular programs and topologies. Specifically, we are interested in studying convergence properties of object migration policies in practice, and in showing that our approach of distributed execution scales to networks with many nodes. There are several other ways of executing ABS programs developed in the HATS project [32], but the main feature we need that is absent from all of them is object mobility between nodes or sites. Also, in contrast with most of these ABS backends, which aim to provide an execution platform for the full ABS language, the simulator only supports a subset of the Core ABS language; notably, the await statement is not supported.

The simulator is implemented in Java. Each node controller is implemented as a Java thread, which communicates with other controllers through TCP sockets, using the Kryonet network library [5]. One reason for choosing to use sockets is to enable to scale simulations over several physical machines and a large number of simulated network nodes. All node controllers in the network have a representation of the abstract syntax tree of the ABS program being executed, which is generated from ABS program code by the lexing and parsing frontend shared by most ABS backends.

As in the conceptual model and the formal semantics, a node controller can have zero or more objects, each having at most one active task. An active task has a reference to the statement currently being executed in the abstract syntax tree. We call an object active if it has an active task. Scheduling of active tasks is done at the node controller level in a round-robin fashion for active objects. More precisely, the scheduler deterministically steps all active tasks, checks for active objects, and then repeats the process on the new set of active tasks.

We implement statement execution by interpretation. The main reason for this choice is to enable easy serialization of objects between executing statements; to get immediate results from load balancing, we must be able to migrate active objects. One drawback of using interpretation is that local execution is slow and resource-demanding compared to the standard ABS backends.

A node controller is associated with a unique TCP port on the host system. Besides a list of neighbour handles, which abstract over underlying sockets, and a list of local objects, the node controller maintains a routing table which is broadcast to neighbours on update. Hence, except after a short interval with many updated locations, we expect routing tables to be up-to-date or nearly so. The node controller also stores incoming messages that cannot be processed locally or rerouted.

Network topology setup and program loading is handled by scripting on top of a custom simple command-line interface (CLI). When starting up, a node controller is assigned a migration policy through the CLI, which is assumed to be the same for all node controllers in the network. A migration policy is based on one of the adaptation strategies described below.

By default, the simulator starts the initial task of the initial object on a single startup node. In all our programs, the initial task creates all the objects used for the duration of the program. Migration and logging does not commence until a method with the name setupFinished is called on some object. There are several reasons for this kind of initialization; it is easier to predict load balancing behaviour with a fixed set of objects, and it is problematic to create new objects on the fly without proper distributed garbage collection, which we have not implemented.
4.5.2 Scenarios

There are many parameters to consider when setting up interesting scenarios for studying adaptation via simulations, as outlined below.

Network configuration The size and topology of the network. Large and dense networks obviously give more overhead in the form of messaging (e.g. routing and load), making simulations slower.

Object behaviour The number of objects generated by the program, intra-object communication patterns, and the fraction of objects with active tasks over time. In practice, this means selecting the appropriate ABS program and adjusting some method parameters.

Adaptation strategy This includes both the logic for deciding when and where to ship away objects, and for messaging to exchange information used as basis for decisions.

By necessity, we can explore only a small cross-section of the possible parameters, at this initial stage of the work.

Network Configurations

The possible sizes of networks to be simulated is limited by the performance of the prototype simulator. Currently, in the order of 25 network nodes can be simulated in reasonable time. On this note, we limit the evaluation to networks with three distinct underlying network topologies for nodes along the continuum from sparsely to fully connected: grids, hypergraphs and full meshes. Our base initial setup for each topology has 32 nodes.

Benchmark Programs

We have developed a number of ABS programs specifically to run in our simulator. We have avoided dynamic object creation to get as few garbage objects as possible; after initial object initialization, the number of objects remains constant. Another important constraint when developing the programs is the need, given a specific network configuration, to reason without too much difficulty about what an optimal allocation of objects to nodes looks like, based on minimizing both load and number of messages exchanged. This means that, for example, the communication graph of objects should be determinable before runtime, and most objects should be active most of the time, ensuring they are eligible for load-based migration. The programs are listed in Appendix A.

IndependentTasks.abs The starting task simply generates objects, and each generated object is called upon to perform a long-running task. There is no communication among workers—only between the coordinator object, which initializes and assigns tasks, and the generated objects. Since there is no communication, an optimal allocation is simply a completely even distribution of objects to nodes, regardless of the network topology.

Ring.abs The starting task generates objects which know the identifiers of the next object in the ring. The last object generated gets the identifier of the first object. The first object, when called, calls its next object, and so on, until the object which has the first object as next object is reached.

Star.abs The starting task generates a certain number of object star configurations, where each consists of one “center” object and one or more “fringe” objects. In each star, the latter objects communicate continually with the former object, but not with each other. There is no inter-star communication at all.
Adaptation Strategies

We have generally restricted ourselves to strategies that as a first priority balance out load evenly among nodes in the network. As a consequence, a simulator node controller continually exchanges load messages with neighbours, regardless of the specific logic for deciding when and where to move objects.

In the simulator, each migration policy defines a callback method which takes the affected node controller as a parameter. The callback method is invoked, and can possibly result in the migration of several objects to adjacent nodes.

Berenbrink et al. An adapted version of the selfish distributed load balancing algorithm by Berenbrink et al., which does not allow neutral moves. One notable difference in the simulator implementation from the abstract description given in Algorithm 1 is that only a fixed small number of objects have the possibility to migrate in each cycle, because of restrictions in the size of message buffers.

Berenbrink et al. with neutral moves An adapted version of the selfish distributed load balancing algorithm by Berenbrink et al., which does allow neutral moves, and therefore is only expected to converge to a completely stable state after a long time, exponential in the size of the network. As determined experimentally, only migrating one or two objects per cycle leads to significantly less oscillation of objects than when directly implementing the abstract description given in Algorithm 2.

Berenbrink et al. with communication intensity A variant of the preceding policy, where the objects selected for migration are selected for their affinity to the (randomly) chosen neighbour node, as determined by their communication history with objects located somewhere in the direction of the neighbour node.

Weighted neighbour load difference Once every cycle, an object and a neighbour node is chosen uniformly at random. Then, a coin biased according to the current load and the chosen neighbour’s load is flipped to decide whether the migration takes place.

Weighted neighbour load difference with communication Similar to the preceding policy, but the coin flip is also biased according to the object’s communication history in direction of chosen neighbour.

Algorithm 1 Berenbrink et al. load balancing cycle

\[
\text{for each active object } o \text{ do}
\]
\[
\quad u' \text{ is a neighbour chosen uniformly at random}
\]
\[
\quad l \text{ is the current load}
\]
\[
\quad l' \text{ is the last known load of } u'
\]
\[
\quad \text{if } l > l' + 1 \text{ then}
\]
\[
\quad \quad \text{send } o \text{ to } u' \text{ with probability } 1 - l'/l
\]
\[
\text{end if}
\]
\[
\text{end for}
\]

Algorithm 2 Berenbrink et al. load balancing with neutral moves cycle

\[
\text{for each active object } o \text{ do}
\]
\[
\quad u' \text{ is a neighbour chosen uniformly at random}
\]
\[
\quad l \text{ is the current load}
\]
\[
\quad l' \text{ is the last known load of } u'
\]
\[
\quad \text{if } l > l' \text{ then}
\]
\[
\quad \quad \text{send } o \text{ to } u' \text{ with probability } 1 - l'/l
\]
\[
\text{end if}
\]
\[
\text{end for}
\]
4.5.3 Scenario Objectives

Since our primary objective is to balance node load evenly, we record the load of all individual nodes over time, and then show maximum load and load standard deviation as proxy measures for how even allocations are. For scenarios with little to no object communication, this is the only measure that is relevant with respect to our objectives. For scenarios with significant messaging, we also consider the number of messages for each node between sampling intervals—with the average number of messages and standard deviation shown as proxy measures. In cases where it is clear that an optimal allocation has all communicating objects placed at one hop’s distance from each other, the number of forwarded messages can act as a proxy measure for progress towards an optimal allocation.

We sample the required quantities from simulations at a fixed global rate, corresponding roughly to a certain number of transitions (1000) in the semantics with imposed fairness via round-robin scheduling. The imposed fairness provides a degree of synchrony in the simulated network.

4.5.4 Results

Simulations of IndependentTasks.abs

The program in total creates 201 objects: one starting object which becomes inactive after initialization and 200 objects that each have a task that runs for the course of the program.

As expected, the algorithm by Berenbrink et al. without neutral moves converged very quickly and stayed unchanged with no migrations, after reaching a state where neighbour load differences have a maximum of one. For most of the runs on a 32-node hypergraph network topology, the stable state coincided with a completely balanced allocation, or very closely so. For the case of a 32-node grid, the stable allocation was frequently some distance from a fully balanced one.

The algorithm variant with neutral moves and two migrations per cycle converges to an almost-stable state quite quickly on a hypergraph, but continues to have minor oscillation of objects. With the same algorithm but five migrations allowed per cycle, there is considerably more oscillation going on after coming close to a balanced allocation. On a grid topology, where a stable allocation can be further away from a balanced allocation, allowing neutral moves gives better results than disallowing them, as expected.

The maximum load of any node and the standard deviation of node load over time for a 32-node hypergraph network topology is shown in Figure 4.3 and Figure 4.4, respectively. The corresponding measures over time for a 8x4 grid network topology are shown in Figure 4.5 and Figure 4.6, respectively. For a grid, the gain from using neutral moves is most distinctly recognized in the lower standard deviation compared to the algorithm without neutral moves in Figure 4.6.

Simulations of Star.abs

In the star program, we construct stars so that each node can hold a whole star, and there is precisely one star per network node. In an optimal allocation, therefore, there are no node-to-node message exchanges at all; all messages are sent locally.

We expected the pure load balancing policies to have markedly worse results than the policies taking inter-object communication intensity into account. In Figure 4.7, the standard deviation of the number of messages sent by nodes is shown. In Figure 4.8, the average number of sent messages over time is shown. In both cases, the measurements have been smoothed out via averaging over ten samples to avoid noise. As can be seen in the figures, there is a distinct improvement with respect to messages sent when using the algorithm by Berenbrink et al. augmented with message intensity comparisons when compared to the other policies, although it is quite far from the optimum. The algorithm using probabilistic weighting of load and messaging seems to improve the most over time, although it performs similarly to the messaging-augmented load balancing algorithm by Berenbrink et al.

With all the tested migration strategies for the hypergraph, load became evenly balanced relatively quickly, similarly to the case on a hypergraph when running IndependentTasks.abs.
When running a ring of around 130 objects on a 32-node grid, there are balanced allocations with four objects per node (possibly five for some) where all objects that communicate are on either the same node or adjacent nodes. The idea is that two of the objects on a node are part of a segment of the ring, while the other two are part of another segment coming back the other way. Such allocations lead to few inter-node messages being needed for a method invocation that involves the whole ring.

In Figure 4.9, the standard deviation of messages of a 129-object ring for a grid topology is shown. Here, both the solutions which take message intensity into account show considerable improvement over time. This is also reflected in the average number of messages sent over time shown in Figure 4.10.
4.6 Conclusions and Future Work

The evaluation suggests that it is feasible in a decentralized setting to meet the objective of balanced resource allocation, and also make headway towards the objective of minimizing communication of distributed objects. However, success for a specific migration strategy depends very much on the scenario it is used in, i.e., the underlying network topology and the specific object communication patterns.

In future work, we plan to continue the theoretical and simulation-based studies to deepen our understanding of multi-dimensional resource management, to improve the performance and accuracy of the simulator, and to investigate adaptation in dynamic networks, initially only with benign churn, i.e., with controlled startup and shutdown of nodes. One problem in doing so concerns the preservation of objects
Figure 4.7: Std. deviation of sent messages for hypergraph in Star.abs

Figure 4.8: Avg. sent messages for hypergraph in Star.abs

Figure 4.9: Std. deviation of sent messages for grid in Ring.abs
in the face of node shutdown. With the ability to safely add and remove nodes to the network comes the possibility to ensure that all the nodes in the network have a load within some given range, while minimizing the number of nodes used.

Figure 4.10: Avg. sent messages for grid in Ring.abs
5.1 Introduction

Various application domains require systems that run continuously and without interruption. This is particularly true in mission-critical applications and any other type of highly-available application. On the other hand, software systems need to be updated in order to keep up with changing operating environments, changing requirements, bug fixes, and expansion or removal of functionality. Such systems need the ability to update their software without interrupting the system’s execution [50].

Typical Software Product Lines (SPL) approaches do not focus on dynamic aspects, and the reconfiguration of products occurs mainly statically at development time [10]. Dynamic Software Product Lines (DSPL) enable a product to be reconfigured dynamically at runtime. Therefore, dynamic software reconfiguration is understood as the ability of reconfiguring products at runtime, that is, the transformation of a product into another valid product without any kind of interruption in the running system. The reconfiguration, in this context, would take place without the need to halt the system, recompile and redeploy. From a technical perspective, dynamic reconfiguration is a challenging task due to reasons such as ensuring that dynamically updated systems will behave correctly [47] or ensuring that no state data is lost [49]. However, even if from a technical perspective things can be changed at runtime, from the Product Line (PL) perspective only certain runtime changes may make sense and preserve the consistency of a software product. Therefore, in order to preserve the consistency of the PL products when reconfigured at runtime, there must be a way to restrict the adaptations that could be performed at runtime. Thus, the adaptations performed at runtime have to be planned in advance, and the resulting product (after the adaptations are performed) must be a consistent product following all feature model constraints and definitions. Therefore, PL reconfiguration poses even more challenges.

We use the ABS modeling framework as the foundation of our solution [30], and the existing support for product line development in terms of static product generation. Static product generation introduced support for configuring a particular SPL product at compile time by taking an ABS core model and a set of delta modules and flattening them to obtain an executable core ABS model of that single product. While static and dynamic product configuration are related concepts, they differ in one key aspect. Static product configuration always starts with the base product (represented by a core ABS model) and applies a sequence of modifications until a product specified by the product line is obtained. Dynamic product reconfiguration starts with any product already configured using the above process, and applies a set of modifications to obtain a new product (out of the set of specified products). The set of products that are configurable from a given product at runtime is constrained in the sense that they have to be explicitly defined and listed. Figure 5.1 illustrates this aspect. To support this kind of dynamic reconfiguration, ABS models need to accommodate dynamic changes in their structure and behavior. Adding this facility to ABS complements the static SPL modeling capability of ABS.

The contribution of this chapter is the added support for runtime product reconfiguration to ABS, obtained by adding a dynamic representation of product specifications and deltas, and by deferring the flattening
process to runtime.

5.2 Related Work

The goal of this section is to describe related work in the literature regarding dynamic software update, DSPL, and also highlight the contributions that are part of the HATS framework and relate to our approach.

5.2.1 Dynamic Software Update

Dynamic software updates and their different characteristics have been widely investigated before and several challenges have been identified and solutions proposed. Hicks and Nettles [50] stated the importance of dynamic updates, particularly for applications such as financial transaction processors, telephone switches, and air traffic control systems, among others.

Multi-threaded software may pose additional challenges to dynamic updates, and Neamtiu and Hicks [67] investigated dynamic updates in this context, focusing on the problem of applying an update in a timely fashion while still producing correct behavior. Hayden et al. [48] also investigated dynamic software updates for multi-threaded software and stated, based on empirical evidence, that update points can be used without creating indefinite delays in the update process.

Another important aspect that has been investigated is the state transfer for efficient updates. State transfer techniques intend to preserve the existing state of the running system, and transfer it to the new system structure after reconfiguration. Hayden et al. [49] investigated the state update field and proposed Ekiden, a state transfer updating library for C/C++ programs. Magil et al. [65] presented a solution to the automatic checking of the state transformation by running tests on the old and new software versions separately and establishing a correlation between old and new-version objects. In a later study, Hayden et al. [47] defined the first methodology for automatically verifying the correctness of dynamic software updates.

Recent work by Wernli [85] proposed an approach using first-class contexts, called Theseus. First-class contexts make global updates unnecessary, since existing threads run to termination in an old context while new threads are started in a new, updated context.

5.2.2 Dynamic Software Product Lines

Typical SPL approaches do not focus on dynamic aspects; if a product needs to be reconfigured, the reconfiguration occurs statically at development time [10]. Dynamic SPL is an emerging field of study, and typically overlaps with other technologies, particularly self-adaptive systems [10]. Most of the work in the DSPL field focuses on managing bounded adaptivity (known and planned in advance) and not unexpected adaptations [10]. Hinchey et al. [52] highlighted the main differences between SPL and DSPL and compared four relevant points, making explicit where the difference is:

- Variability management in SPL describes different possible systems, and variability management in DSPL describes different adaptations of the system.
• The reference architecture in an SPL provides a common framework for a set of individual product architectures, and a DSPL architecture is a single system architecture, which provides a basis for all possible adaptations of the system.

• Business scoping in SPL identifies the common market for the set of products, and adaptability scoping identifies the range of adaptation the DSPL supports.

• The two-life-cycle approach in SPL describes two engineering life cycles, one for family engineering and one for application engineering. The DSPL engineering life cycle aims at the systematic development of the adaptive system, and the usage life cycle exploits adaptability in use.

Damiani and Schaefer [28] proposed a delta-oriented DSPL, with a reconfiguration automaton specifying how to switch between different feature configurations. Damiani et al. [27] also provided a formal foundation for delta-oriented DSPL.

5.2.3 Related Work within HATS

Our work has some relations with a number of other contributions inside the HATS project and those related contributions are described next.

Deliverable 3.3 - ABS Component Model

This section describes the component model extension contribution detailed in Deliverable 3.3 [31]. The goal of the component model extension is to enable dynamic reconfigurations by adding the component notion on top of the ABS objects to enhance objects and object groups with the basic elements of components (ports, bindings, consistency, and hierarchy). The enhancements proposed were: the notion of an output port distinctive from the object’s fields; the keyword Critical to annotate methods to represent that while an instance of the method is executing, it is not in a safe state to be updated; a primitive to wait for an object to be in a safe state before update; and the hierarchy of locations where components can move within the location hierarchy.

Some overlap exists between our proposal and the component model previously discussed. To avoid any duplicate effort we first analyzed the aforementioned component model before designing our contribution.

Deliverable 3.3 - Dynamic Modeling of Product Lines

An overview and formalization of dynamic product lines in the context of abstract delta modeling, was also provided in Deliverable 3.3 [31]. In this context, the DSPL takes the form of a Mealy Machine, a finite automaton with an input symbol and an output symbol on every transition. The input symbol corresponds to a feature that has been turned on or off and the output symbol corresponds to the delta that has to be applied to the current product to bring it up to date. Based on this representation of dynamic product lines, a cost model was also introduced. Monitoring a specific feature for change has a certain cost, and some features are more costly than others. Then, it was described how to optimize dynamic product lines by selectively removing transitions from them, effectively disregarding costly features until they become relevant. More details can be found in Deliverable 3.3 [31].

Deliverable 3.3 - MetaABS

To cope with the need to modify ABS code at runtime, KUL introduced a new reflective layer that allows introspection and manipulation of running code. This layer is exposed in a language extension called MetaABS (more details in Deliverable 3.3 [31]).

The purpose of MetaABS is to provide a unified interface for various runtime model analysis tasks. Adding meta-programming capabilities to ABS means that certain model analysis tasks can be encoded in ABS and carried out automatically while the model is executing. Meta-programming is generally understood as the
ability to observe and modify the structure and behavior of a program from within a program, either statically or at runtime. A meta-programming interface exposes basic elements of the programming language and the runtime environment to the programmer, enabling their inspection and modification. While it exposes these elements, it also abstracts away from their implementation.

Languages that support meta-programming commonly achieve this by providing reflection, that is, the ability of a program to inspect and modify itself at runtime. Thus, the meta-program (the program-transforming program) and the program that is transformed are the same. Reflection is decomposed into introspection, meaning the ability of a program to examine itself, and intercession, which enables a program to modify its state and behavior. In other words, introspection and intercession provide read and write access, respectively, to elements of the language. For example, the Java Reflection API is a meta-programming interface that provides methods for examining, and to a very limited extent, modifying the runtime properties of objects including their class, interfaces, fields, and methods.

MetaABS is used in the context of our work to enable runtime reconfigurations, and the details of how MetaABS is used in this context are described in Section 5.5.1.

Deliverable 3.6 - Evolvable Systems

The summary of the work conducted in HATS WP 3 is presented in Deliverable 3.6 [33]. The goal of Deliverable 3.6 is to provide only a summary of the work conducted in other deliverables. Some topics are described, but not in depth (i.e., MetaABS is briefly presented in Deliverable 3.6, but it is presented in detail in Deliverable 3.3 [31]).

5.3 Static Variability in ABS

ABS is an object-oriented language designed to formally specify large software systems. The core of the ABS language resembles standard programming languages like Java, with concurrency and functional constructs. Besides this core language, four other elements exist in ABS to model and analyse variable systems following SPL [72] engineering practices. This work focuses on these four elements, enumerated below (together with the core module), and how they can be extended to model dynamic product reconfiguration.

**Feature model** This is used to model all products of an SPL by using features and feature attributes [21].

**Product selection** A product selection identifies an individual product that is of particular interest, defined by a valid combination of features.

**Core module** The core module consists of the classes that implement a complete product of the corresponding product line. Typically, the core module represents one product configuration, from which other products are derived (with the application of deltas).

**Delta module** This is a reusable unit of ABS code that can be applied incrementally to an ABS model to adapt its behaviour to conform to a particular product.

**Configuration** A configuration associates features to delta modules, enabling us to generate the ABS model for an individual product by naming it.

Throughout this chapter, we will explain these elements using a running example of a Chat product line, which is targeted at creating different chat clients.

5.3.1 Feature model

A feature model is described as a feature diagram, which is a tree of features with associated attributes and cardinality restrictions [77]. In our Chat product line example, the variations consist of the chat mode (textual, via voice, or via video), and the possibility of transferring files or not.
A feature diagram describes what combinations of features are valid. In our example, the diagram of the Chat product line is depicted in Figure 5.2: there must be at least a chat mode feature selected and the FileTransfer feature is optional. The two implications below the diagram capture extra constraints, namely that the Voice feature requires Text to be selected, and Video requires Voice. Our textual notation of feature diagrams and the usage of attributes are left out of this report since they do not play a role in the extension for dynamic reconfiguration of products.

5.3.2 Product selection

The syntax for describing product selections allows the enumeration of valid products from the product line. In our case, Figure 5.3 declares three products: HighEnd, Regular, and LowEnd. To produce the product Regular, for example, the features Voice and Text are selected, and the final code is generated based on the delta modules, as described in the following subsections.

```
product LowEnd (Text);
product Regular (Voice, Text);
product HighEnd (Video, Voice, Text, Files);
```

Figure 5.3: Example of product selections

5.3.3 Core module

The core module, or core for short, defines the set of classes and interfaces that are part of a product configuration, without the need to apply any deltas. In our case, in Figure 5.4 the core module is composed of the classes that are needed in order to implement a LowEnd chat product (that only supports textual chatting as defined in the product selection in Section 5.3.2).

5.3.4 Delta module

A delta module, or delta for short, associates a name to a set of program transformations, as illustrated in Figure 5.5. These transformations are with respect to a core ABS program, although we will explore later how to describe transformations with respect to other products. The details of how the program transformations are specified are not relevant for this chapter. These program transformations express the addition, removal, or replacement of classes, interfaces, functions, methods, and fields.
interface Client {
    ...
}

interface Text extends Client {
    Unit message(Client client, String msg);
    // other method definitions
}

class ClientImpl implements Client, Text {
    Unit message(Client client, String msg) { ...
    ...
}

Figure 5.4: Definition of core module classes and interfaces

delta DVoice {
    // modifications to the core to add voice functionality
}
delta DVideo {
    // modifications to the core to add video functionality
}

Figure 5.5: Definition of delta modules

5.3.5 Configuration

A configuration is a sequence of statements, each associating a delta to a set of features. In the example presented in Figure 5.6, the delta DVideo is associated to the Video feature, meaning that when the Video feature is selected, the DVideo delta should be applied. In fact, the set of features preceding the when keyword is a Boolean formula over features that is verified against the feature selection.

productline ChatPL;
features Text, Video, Voice, Files;
delta DVideo when Video;
delta DVoice when Voice;

Figure 5.6: Configuration of the Chat product line

5.4 Modeling DSPLs

ABS was designed from the beginning with variability support in mind and includes first-class constructs dedicated to modeling variability (such as feature models and deltas, cf. previous Section 5.3). Variability modeling was initially limited to the static level, as it was removed upon compilation. We introduce support for dynamic SPLs (DSPLs), enabling product reconfigurations at runtime. To enable DSPL models, we extend the ABS variability constructs in a conservative manner, i.e., without affecting existing SPL modeling semantics. This section describes the ABS language concepts and constructs that assist dynamic SPL engineering.

In Figure 5.7, the main aspects of the dynamic support for ABS are highlighted. Some existing aspects needed to be updated (marked in green) and new concepts were created (marked in blue). Inside the runtime circle in Figure 5.7, the arrows represent the following transitions:

From trigger to reconfiguration A trigger routine is responsible for checking periodically, based on certain monitoring variables, whether a reconfiguration should be performed or not. In this stage, the
trigger has access to all the available reconfigurations of the products (the reconfiguration logic is described in Section 5.4.1). Using our ChatPL example, in Section refsec:BuildingSelf-adaptingSystems, we will describe how this checking takes place.

**From reconfiguration to reconfigured product** MetaABS is the key actor at this point, performing the reconfiguration by applying the specific deltas defined in the DSPL configuration (described in Section 5.4.2) and applying the state transfer definitions (described in Section 5.4.3).

**From product to trigger** Each product configuration must have a trigger routine that should be periodically triggered in order to check possible reconfigurations.

These transitions will be discussed in more detail in Section 5.5.

---

**Figure 5.7: Dynamic aspects of ABS DSPLs**

---

5.4.1 Product selection

A DSPL in ABS is a set of software products that are available at runtime together with an initial product. The initial product is the product that has been configured statically and is active when the system is deployed and running. When the initial product is reconfigured dynamically, a different product becomes active and the system behaves according to the specification of the new product. That new product can be reconfigured into yet another product and so forth.

ABS requires explicitly declaring the possible dynamic transitions between products. Products of static SPLs are selected in ABS simply by associating a product name with a set of features (cf. Figure 5.3). For dynamic SPLs, the product declaration additionally lists the other products of the SPL that the given product can be transformed into at runtime.

**Syntax** A product selection includes an optional set of Adaptations. An Adaptation denotes a product into which the current product can be transformed. The by clause specifies the state update function that needs to be applied when doing this particular product transformation. How a product is to be transformed, that is, the sequence of deltas that need to be applied to the current product, is determined – as in the static context – from the product line configuration. Figure 5.8 presents the product selection grammar.
Syntax:

\[
\begin{align*}
\text{Product Selection} & ::= \text{product} \ 	ext{TypeId} \ ( \text{FeatureSpecs} \ ) \ \{ \ [ \text{Adaptation}^* ] \} \\
\text{FeatureSpecs} & ::= \text{FeatureSpec} \ ( , \text{FeatureSpec} )^* \\
\text{FeatureSpec} & ::= \text{FID} \ [ \text{AttributeAssignments} ] \\
\text{AttributeAssignments} & ::= \{ \text{AttributeAssignment} \ ( , \text{AttributeAssignment} )^* \} \\
\text{AttributeAssignment} & ::= \text{AID} = \text{Literal} \\
\text{Adaptation} & ::= \text{TypeId} \ \text{by} \ \text{Update} ; \\
\text{Update} & ::= \text{TypeId}
\end{align*}
\]

Figure 5.8: Product selection grammar

Example  Figure 5.9 shows a product selection example that enables runtime reconfiguration. The “low-end” chat software product implements only the Text feature. This product can be reconfigured at runtime into a “regular” chat product that additionally supports the Voice feature. The third product is a “high-end” chat system that also supports video and file transfer. Updating a system at runtime requires the transition of the system’s execution state to match the updated system structure. In ABS, state transitions are defined using state update declarations, which will be described later in Section 5.4.3.

\begin{verbatim}
product LowEnd (Text) {
    Regular by StateLowToReg;
}

product Regular (Text, Voice) {
    HighEnd by StateRegToHigh;
    LowEnd by StateRegToLow;
}

product HighEnd (Text, Voice, Video, Files) {
    Regular by StateHighToReg;
}
\end{verbatim}

Figure 5.9: Product declarations for the dynamic chat system SPL

The possible runtime reconfiguration paths between products of the Chat product line are illustrated in Figure 5.10 and contrasted to the static configuration options.

Figure 5.10: Static Chat product configuration (left) and dynamic Chat product reconfiguration (right)

5.4.2 DSPL Configuration

Transforming a software product with a certain set of features into a product with a different set of features requires changing its structure and behavior. For an ABS model, this entails adding, removing, or modifying model elements such as classes, functions, and data types. As in the static setting, this is done by applying a set of deltas in sequence, except that now deltas are applied while the system is running. The sequence of deltas that needs to be applied is determined based on the product line configuration.
Compile-time product configuration always starts with a program core and applies a sequence of delta modules to that core. In ABS, this sequence is determined by the product line configuration. A configuration links a feature model, which describes the structure of an SPL, to deltas, which implement its behavior. Features and deltas are associated through application conditions, which are logical expressions over the set of features and attributes in a feature model. The collection of applicable deltas is given by the application conditions that are true for a particular feature and attribute selection.

Syntax The grammar of delta clauses has been extended to include from application conditions. These are evaluated in the context of the current product, while the to (or when) application conditions are evaluated in the context of the target product. We only include the relevant portions of the grammar below. For the full ABS product line configuration grammar, we refer to the ABS reference manual [1]. Figure 5.11 presents the product configuration grammar.

Syntax:

\[
\begin{align*}
\text{DeltaClause} &::= \text{delta} \hspace{2em} \text{DeltaSpec} \ [\text{AfterCond}] \ [\text{FromAppCond}] \ [\text{AppCond}] \ ; \\
\text{FromAppCond} &::= \text{from} \hspace{2em} \text{BoolExp} \\
\text{AppCond} &::= ( \text{when} \mid \text{to} ) \hspace{2em} \text{BoolExp} \\
\text{BoolExp} &::= \text{BoolExp} \hspace{2em} \&\& \hspace{2em} \text{BoolExp} \\
&\quad | \hspace{2em} \text{BoolExp} \mid\mid \hspace{2em} \text{BoolExp} \\
&\quad | \hspace{2em} \sim \hspace{2em} \text{BoolExp} \\
&\quad | \hspace{2em} ( \text{BoolExp} ) \\
&\quad | \hspace{2em} \text{FID}
\end{align*}
\]

Figure 5.11: Product configuration grammar

Example Figure 5.12 shows a product line configuration example for the Chat SPL that governs both static and dynamic (re)configuration. Lines 4–5 apply to the static configuration: the delta DVoice is applied to the core when a product includes the voice feature, and DVideo is applied when the video feature is included.

```
productline ChatPL;
features Text, Video, Voice, Files;
delta DVoice when Voice;
delta DVideo when Video;

// deltas used for runtime adaptation
delta DVoice from Text \&\& \sim Voice \&\& \sim Video when Voice;
delta DVideo from Text \&\& Voice \&\& \sim Video when Video;
delta DNoVideo from Text \&\& Voice \&\& Video when Text \&\& Voice \&\& \sim Video;
delta DNoVoice from Text \&\& Voice \&\& \sim Video when Text \&\& \sim Voice \&\& \sim Video;
```

Figure 5.12: Configuration with deltas used at runtime

With dynamic SPLs, a system that has been configured to resemble a certain product can be re-configured, at some point after its deployment, to behave like a different product. This means that the starting point of the dynamic reconfiguration process can be any product, not just a bare-bones core product. To accommodate this fact, the scope of delta clauses in ABS has been extended to allow two application conditions: one that specifies the feature selection after reconfiguration, introduced by the when (or to) keyword and, additionally, one that is evaluated against the current feature selection (the product before reconfiguration). This additional condition is introduced with the from keyword. The delta clauses in lines 8–11 exemplify
the use of both application conditions. In line 8, the from application condition (Text && Voice && Video) restricts the runtime application of delta Voice to products that have the Text feature but not Voice and Video. The when application condition (Voice) ensures that delta Voice is only applied if the target product will have the Voice feature.

5.4.3 State Transfer

In SPL engineering, configuring a system to behave like a certain product of an SPL involves assembling its code from reusable artifacts such as code modules, patches, or, as in the case of ABS, deltas. This is also true of dynamic SPLs, where behavioral units are available at runtime and can be enabled or disabled at runtime. A running system, however, also has an execution state, i.e., the collection of values assigned to variables and fields, which is commonly stored on a stack and heap. When reconfiguring a running system, these values need to be preserved or adapted to the system’s new structure according to user-defined state transformation directives. ABS manages the transformation of states upon reconfiguration using state updates.

A state update specifies how to transfer the state of objects when transitioning the system to represent a different product of the SPL. A state update has an optional pre and a post block that are executed, respectively, in the system’s pre-transformation state and in its post-transformation state. In our chat example, in Figure 5.13, two state updates are defined for two different transitions (from a regular chat product to a high-end product, and from a high-end to a regular chat product). In this example, a manipulation concerning the audio stream used is performed.

```plaintext
update StateRegToHigh;
uses Chat;
update CallImpl {
  pre: AudioStream tmp = this.singleAudioStream;
  post: this.avStreamManger.includeAudio(tmp);
}
update StateHighToReg;
uses Chat;
update CallImpl {
  pre: AudioStream tmp = this.avStreamManger.extractSingleMainAudio();
  post: this.singleAudioStream = tmp;
}
```

Figure 5.13: State update example

In Figure 5.9 it is possible to visualize which state update is run depending on the reconfiguration being performed.

5.5 Dynamic Reconfiguration

While the previous section introduced the language for modeling DSPLs, this section describes the mechanisms that enable dynamic reconfiguration of executing ABS models. This ranges from description of the runtime representation of the variability model via the reconfiguration process based on dynamic delta application and transfer of state to ABS’s reflective interface, which gives the user control over the reconfiguration process. This section also discusses problems and challenges that arise with runtime adaptation, such as managing processing contexts and dealing with state references that are no longer available.
5.5.1 Building Self-Adapting Systems

Systems that adapt their behavior at runtime often need to do so autonomously, by monitoring certain variables in their operating environment and adapting as these variables change. A common way to allow a system to inspect itself and its environment, and change its own behavior, is through reflection, see Section 5.2.3.

ABS includes MetaABS, a general-purpose meta-programming interface that reflects various aspects of the runtime back to the ABS program, including operations that modify these aspects. Among these aspects are the DSPLs that the system is part of and the capability to reconfigure the system into a different product of the product line.

MetaABS provides a Productline interface that gives access to operations related for querying and adapting the running software product (Figure 5.14). The Productline provides ABS methods to query the name of the current product, the products that can be obtained through reconfiguration of the current product, and an operation to actually reconfigure the current system to behave like a new product. The Productline object itself is obtained by calling the MetaABS function getPL.

We illustrate the usage of the Productline API in Figure 5.15 with the Chat product line example. The reconfiguration logic is encapsulated in a separate class that implements a Reconfigurator interface. A reconfigurator instance will run in a separate cog (i.e., concurrently to the chat functionality), monitor the network connection, and transform the running product depending on the available bandwidth.

```java
class Reconfigurator(Connection conn) {
    Unit run() {
        ProductLine pl = getPL();
        while (True) {
            String p = pl.getCurrentProduct();
            Bandwidth bw = conn.checkBandwidth();
            if (p == "RegularChat") {
                if (bw == Low) {
                    pl.configureProduct("LowEndChat");
                } else if (bw == High) {
                    pl.configureProduct("HighEndChat");
                }
            } else if (p == "HighEndChat") {
                if (bw == Low bw == Mid) {
                    pl.configureProduct("RegularChat");
                }
            }
        }
    }
}
```

Figure 5.15: Implementing runtime product reconfiguration for the Chat product line using MetaABS
5.5.2 Self vs. Externally Triggered Adaptation

In the previous section, we showed how ABS models are capable of auto-reconfiguration: by using ABS’s reflective interface, the running system itself monitors certain aspects of its operating environment and autonomously reconfigures itself. Yet sometimes, changing a system’s configuration is triggered by external decisions, such as the business decision to add or remove certain features. There is no inherent obstacle to allowing ABS models to be reconfigured based on such external triggers. We have experimented with a runtime interface that receives reconfiguration directives and executes these using the same reconfiguration primitives that are accessible through the reflective MetaABS API. Another approach would be to include such an interface as a feature of the ABS system and implement it in ABS by forwarding reconfiguration commands to the reflective API.

5.5.3 Adaptive Runtime Environment

The ABS language is accompanied by a comprehensive tool framework designed for developing and analyzing various aspects of ABS models [86]. The ABS compiler includes several back ends that generate code in different formats, some more suitable for formal analysis, others more suitable for producing portable, executable code. In conjunction with extending the ABS language to support DSPL modeling, we developed the ABS dynamic Java back end. The dynamic Java back end generates standard Java code. Using the built-in compiler of the ABS tool chain, it directly compiles an ABS model to JVM bytecode. DSPL models generated using the dynamic Java back end are capable of dynamic reconfiguration while executing on the JVM.

The key idea is to use dynamic structures in the target language to represent ABS language elements. The dynamic Java back end uses Java (singleton) objects to represent interfaces, classes, methods, objects, object fields, deltas, products, etc. Such a representation trades execution performance for fully malleable ABS models.

5.6 Conclusion

The demand to run systems continuously without interruption is increasing every day. In addition, the need to quickly respond to changes according to changing requirements/environments/conditions is also a vital need and has been gaining a lot of attention. The absence of such capabilities forces companies to follow different approaches in order to adapt their products quickly, and with the minimum interruption time possible. However, such approaches are typically very effort-intensive and error-prone.

In DSPLs, the main goal is to explore the already existing variation inside the products in order to support adaptations at runtime. DSPLs provide an interesting setting to apply the concept of dynamic updates by taking advantage of the existing variation. In this context, being able to explore existing variability at runtime requires the adaptations to be carefully planned and defined. Besides, product consistency according to the feature model (after reconfiguration) is a major concern.

Therefore, the contribution of this task was to extend the static capabilities of the ABS language and SPL support to enable dynamic aspects in the SPL development aligned with the concept of DSPLs. The Product Selection was extended in order to include also the definitions of possible reconfigurations, and the Product Configuration was extended to establish relations among the reconfigurations and the deltas. Besides, the concept of state transfer updates was defined, with the intent to preserve the states of the running products. To support the dynamic application of deltas, MetaABS was also extended and used as the foundation of the dynamic reconfiguration.
Chapter 6

Conclusion

In this deliverable we have studied autonomous evolution of software system from two perspectives:

1. From the point of view of performance adaptation, where the task is to fit a statically known software configuration to a network such that (1) functional behaviour is unaffected, and (2) various performance parameters such as processor load or communication “intensity” are optimized.

2. From the point of view of functional adaptation, where the task is to reconfigure a running system without having to recompile or redeploy the system.

Many issues remain to be resolved. We start by discussing performance adaptation and ABS-NET.

As the model stands we only study adaptation of a static software system to a static processor configuration. Approaches to handle dynamically evolving software are proposed in several HATS deliverables, including the present, and these are to a large extent orthogonal to the performance adaptation issues studied in ABS-NET. To support dynamically changing execution platforms, such as dynamically changing network configurations, more work is needed. In preliminary work not reported here we have developed a model of benignly dynamic networks, essentially to study power adaptation by node shutdown/power up. Beyond this, crash failures and byzantine failures deserve to be studied in an ABS-NET context in the future.

Within the HATS project the notion of deployment component [58, 59, 57] has been proposed in D2.1 as an approach to make resource limitations reflecting, e.g., time or processor capacity, available at the source code (ABS) level. Usually, these resource limitations are modeled in the abstract, without tying to real time, or real processor utilization, which has the advantage that these quantities can be measured and acted upon to an explicitly programmable level of precision. Reflection then becomes available to program resource management strategies directly in ABS itself. The same type of reflection could be done for ABS-NET as well, and preliminary work at UiO and KTH indicates that this is a potentially fruitful avenue for further research.

A further interesting topic is the practicality of the ABS-NET model. Even if the model itself is strongly motivated by practical, cloud-like scenarios (collections of processes executing transparently on top of a processor network) and could (and should) in principle be implemented on top of a cloud infrastructure, we are as yet only able to address a small part of the performance tuning spectrum in practice. At this point we have really only studied performance tuning by object (VM) migration. Other parameters are equally important, such as adaptive processor scheduling, buffer management (not least!), and processor and channel bandwidth constraints, to name just a few. How to automatically tune performance in such a highly dimensional parameter space in a fully decentralized fashion is a problem we have only managed to scratch the surface of, and it remains one of major research challenges in the area of systems and network management.

In the area of functional adaptation we have demonstrated the applicability of one tool (MetaABS) developed within HATS, to support controlled dynamic software adaptation within a software product line (SPL) context, and a small supporting language to define product selection strategies has been developed. This work indicates one possible approach for runtime software evolution, by using reflection to examine SPL...
constraints at runtime, and to reconnect program deltas. The approach leverages existing SPLs and exploits their variability to support a similar variability within having to resort to reconfiguring or rebuilding the system, but because of this the approach is also limited to changes that are *foreseen* when the SPL was built. The challenge remain how to support *unforeseen* changes, or changes when the system under consideration is only partially known.
Bibliography


Glossary

Terms and Abbreviations

**ABS** Abstract Behavioral Specification language. An executable class-based, concurrent, object-oriented modeling language based on Creol, created for the HATS project.

**ABS-NET** A model for transparently executing ABS programs on a static network of processors.

**Adaptation** The process modifying a system or a piece of software to make it function better, faster, more safely, more securely, or with less resource consumption.

**Barbed equivalence** A notion of equivalence of programming language terms based on a rewrite relation with observability predicates defined on the terms.

**COG** Concurrent Object Group, the unit of parallelism in ABS.

**Core ABS** The behavioural functional and object-oriented core of the ABS modeling language.

**Compiler back end** The functional entity of a compiler that is mainly concerned with generating code for a specific machine.

**Delta** Synonymous with delta module.

**Delta module** A specification of modifications to core ABS language elements (classes, methods, interfaces, etc.)

**DHT** Distributed Hash Table. Distributed data structure commonly used in modern peer-to-peer networks for fast search over an unstructured index set.

**Dynamic software product line (DSPL)** A set of software products that can be adapted dynamically by adding and removing features.

**Feature** Generally, an increment in software functionality. On the level of feature models it is merely a label with no inherent semantic meaning.

**Feature model** An expression of the variability within product lines. Abstractly it may be seen as a system of constraints on the set of possible feature configurations.

**IDE** Integrated Development Environment.


**Network semantics** A semantics of a programming language given as an execution model on a message passing network of processing nodes.

**NID** Node identifier.
OID  Object identifier.

Routing  The process of determining a path for a message to take in order to find its way from one node in a network to another.

RPC  Remote Procedure Call

Scheduling  The act of choosing one of a set of processes for execution.

Software component  A modelling abstraction reflecting the logical units of composition, which provides isolation, mobility, and data-flow reconfiguration capacities.

Software product  A software systems with a well-defined set of features.

Software product reconfiguration  The process of adding and removing features from a software system at runtime.

Software product line (SPL)  A set of software products that share a number of core properties, and differ on other aspects.

Software product line engineering  A development methodology for software product lines.

VM  Virtual machine
Appendix A

Adaptibility-Simulated Core ABS Programs

A.1 IndependentTasks.abs

```abs
module Load;
interface Load {
    Unit createLoad();
}
interface Util {
    Unit setupFinished();
}
class LoadImpl implements Load {
    Unit createLoad() {
        Int i = 0;
        while (i < 12000) {
            i = i + 1;
        }
    }
}
class UtilImpl implements Util {
    Unit setupFinished() { }
}

{ Int i = 0;
  while (i < 200) {
    Load task = new LoadImpl();
    task!createLoad();
    i = i + 1;
  }
  Util util = new UtilImpl();
  util!setupFinished();
}
```

A.2 Ring.abs

```abs
module Ring;
```
interface RingNode {
    Int aggregate();
    Int aggregateCaller(RingNode caller);
    Unit setFollower(RingNode follower);
}

interface RingBuilder {
    RingNode buildRing(Int lengthAfterInitial);
    Unit setupFinished();
}

class RingNode(Int val, RingNode follower) implements RingNode {
    // assume ring is completed
    Int aggregate() {
        Fut<Int> fAggregate = follower!aggregateCaller(this);
        Int i = 0;
        while (i < 5) {
            i = i + 1;
        }
        Int aggregate = fAggregate.get;
        return val + aggregate;
    }

    // assume ring is completed
    Int aggregateCaller(RingNode caller) {
        Int aggregate = val;
        if (follower != caller) {
            Fut<Int> fFollower = follower!aggregateCaller(caller);
            Int i = 0;
            while (i < 10) {
                i = i + 1;
            }
            Int aggregateFollower = fFollower.get;
            aggregate = aggregate + aggregateFollower;
        }
        return aggregate;
    }

    Unit setFollower(RingNode node) {
        this.follower = node;
    }
}

class RingBuilder implements RingBuilder {
    // assume lengthAfterInitial >= 0
    // ensure all follower identifiers are set
    RingNode buildRing(Int lengthAfterInitial) {
        RingNode initNode = new RingNode(0, null);
        RingNode prevNode = initNode;
        Int currLength = 0;
        while (currLength < lengthAfterInitial) {
            RingNode currNode = new RingNode(currLength + 1, prevNode);
            Fut<Unit> fFol = currNode.setFollower(prevNode);
            //Unit u = fFol.get;
prevNode = currNode;
currLength = currLength + 1;
}
Fut<Unit> fFolInit = initNode!setFollower(prevNode);
Unit u = fFolInit.get;
return initNode;
}

Unit setupFinished() { }

{ }

RingBuilder builder = new RingBuilder();
RingNode fst = builder.buildRing(128);
builder!setupFinished();
while (True) {
    fst!aggregate();
    Int i = 0;
    while (i < 40) {
        i = i + 1;
    }
}

A.3 Star.abs

module Star;

interface StarBuilder {
    Unit buildStars(Int stars, Int fringes);
    Unit setupFinished();
}

interface Center {
    Int value();
}

interface Fringe {
    Unit query();
}

class Center(Int val) implements Center {
    Int value() {
        Int i = 0;
        while (i < 2) {
            i = i + 1;
        }
        return val;
    }
}

class Fringe(Center c, Int val) implements Fringe {
    Unit query() {

while (True) {
    Fut<Int> f = c!value();
    Int i = 0;
    while (i < 10) {
        i = i + 1;
    }
    Int result = f.get();
}

class StarBuilder implements StarBuilder {
    Unit buildStars(Int stars, Int fringes) {
        Int i = 0;
        while (i < stars) {
            Center c = new Center(i);
            Int j = 0;
            while (j < fringes) {
                Fringe f = new Fringe(c, i);
                f!query();
                j = j + 1;
            }
            i = i + 1;
        }
    }
    Unit setupFinished() {
    }
    {
        StarBuilder builder = new StarBuilder();
        builder.buildStars(32, 6);
        builder!setupFinished();
    }
Appendix B

Paper 1: Location Independent Routing in Process Network Overlays
Location Independent Routing in Process Network Overlays

Mads Dam
School of Computer Science and Communication
KTH Royal Institute of Technology

Abstract
Location transparency, i.e. the decoupling of objects, applications, and VM’s from their physical location, is a highly desirable property, for instance to aid application development, to help management and fault isolation, and to support load balancing and efficient resource allocation. Building location transparent systems on top of ip requires objects to be addressed in terms of their hosting node. When objects migrate their hosting node changes, creating a need for a message routing infrastructure using e.g. location servers or message forwarding chains. Such an infrastructure is costly in terms of complexity and overhead. By instead using location independent routing, it is possible to direct messages to the receiving object instead of that objects location, which may be out of date. We show how this idea allows complex object overlays to be implemented in a sound, fully abstract, and efficient (lock-free) manner on top of an abstract network of processing nodes connected by asynchronous point to point communication channels.

1 Introduction

The decoupling of applications from their physical realization, intimately tied to the concept of virtualization, is a recurrent theme in the history of computing. Running applications on virtual machines allows many tasks to be performed independently of the physical computing infrastructure. By migrating virtual machine images between physical processors it is possible for a cloud provider to adapt processing and communication load to changing application demands and to changes in the physical infrastructure. In this way applications can, at least in principle, be freed of the burden of resource allocation. That is, it is left to an underlying processing network to determine on which nodes to place which tasks in order to make efficient use of processing resources, while at the same time meeting requirements on response times and processing capacity. If realized, the result is simpler application logic, better service quality, and, ultimately, lower costs for development, operations and management.

The question is how to realize this potential with a minimum of overhead, and in such a way that applications behave in a predictable manner.
We examine this question in the context of a rudimentary distributed object language, and propose a formal, networked, runtime semantics of this language with some quite novel features. The goal is a “bare-metal” style of semantics where all aspects of computing and communication are accounted for in terms of local operations that could be directly implemented on top of silicon or, say, a hypervisor such as Xen [3].

A key problem is how to handle object and task mobility in an efficient manner. Since the allocation of objects to nodes is dynamic, some form of application level routing is needed to ensure that messages reach their destinations quickly, and with minimal overhead. Various approaches have been considered in the literature (cf. [36] for a survey):

- One option is to maintain a centralized or distributed database of object locations. Such a database can be used for both forwarding, by routing messages through the forwarding server, and for location querying, by using the database to look up destination object locations. In either case, object location and the location database must be kept consistent, which requires synchronization. Many experimental object mobility systems in the literature use some form of replicated or distributed location databases, cf. [13, 36, 4, 19].

- Another option is for nodes to maintain forwarding pointers, as in the Emerald system [24]. Migration then causes forwarding pointer chains to be extended by one further hop, and some mechanism is typically used to piggyback location update information onto messages, to ameliorate forwarding chain growth. This mechanism is used, for instance, in JoCaml [9].

- Many solutions involve some form of broadcast or multicast search. For instance, an object may use multicasting to find an object if a pointer for some reason has become stale, as in Emerald, or for service discovery, as in Jini [1].

- Other solutions have been explored too, such as tree-structured DNS-like location directories [41], Awerbuch and Peleg’s distributed directories [2], and Demmer and Herlihy’s arrow protocol [12].

The main source of the difficulties these approaches are designed to solve is the distinction between destination host identifiers (location) and search identifiers (object identity). But, in a fully mobile setting the location at which an object resides has no intrinsic interest\(^1\). What is of interest is message destination, that an rpc destined for the object with identity \(o\) is routed to the location where \(o\) resides, and not somewhere else. The address of the destination is not relevant. In other words, rather than routing messages according to ip address, inter object message routing should really be done according to the identity of the destination object rather than an assumed host identity, which might for all the sender knows be completely out of date.

\(^1\)Location has interest as a source of latency, for instance, but that is another matter.
This problem is in fact well known in the networking community, and has been the subject of significant attention over the last decade. The idea is to replace the location-based routing of traditional ip networks with location independent schemes that route messages according to names, or content. Names can be flat, unstructured identifiers, as in [6], or they can encode some form of signed content identity, as in content centric networking [20]. The general goal is to devise routing schemes for flat name spaces that are compact, such that routing tables can be represented at each node using space sublinear in the size of the network, and such that path lengths, and hence message latencies, do not grow too far from the optimal. The latter requirement of low stretch, defined as the ratio of route length to shortest path length, precludes the use of both location registers and hierarchical ip-like naming schemes.

The main purpose of this paper is to show that name-based routing offers a new space for solutions to the object mobility problem with some attractive properties:

• No centralized or decentralized object location database is needed, since the network routing mechanism itself ensures that messages are routed to their proper hosts

• A whole swathe of software becomes superfluous, which manages address lookups, message forwarding, rerouting, address bookkeeping, and the synchronization overhead between location registers and the migrating objects is eliminated. As a result, the “trusted computing base” of the networked execution platform is significantly reduced, in terms of size, and complexity

• Traffic overhead is decreased. First, mobility support on top of IP needs to perform routing both at IP and at application level. Name-based routing in effect eliminates the need for IP level routing. Moreover, in steady state the simple distance vector routing scheme used here has stretch 1, so message delivery overhead is minimal (however, distance vector routing is not compact\(^2\), so our scheme does not scale well. We leave this issue for future work).

• Improved robustness: In faulty situations, if connection to the location register is lost, message delivery is impossible (or needs to resort to more costly mechanisms such as broadcasting, as in Emerald or Jini). Routing can be made self-stabilizing and thus able to adapt to any type of disturbance, as long as connectivity is maintained. This allows computation to progress (including delivery of messages and migration of objects) even when the network is under considerable churn

On the other hand, throwing away network layers 3 and above may seem an excessive price to pay, and the argument above that the cost of ip routing should be taken into account is evidently invalid if ip routing needs to be performed anyway in any realistic implementation. We argue, however,

\(^2\)Routing tables in distance vector routing have size $\Omega(n \log n)$
that this does not need to be the case. First, as we have noted, the general architecture of the future internet is currently very much in flux. Second, although we have so far only explored an early prototype simulator [10] built on top of an ip-based overlay it is perfectly possible today to build large scale non-ip networks with only layer 2 connectivity, sufficient to bootstrap our approach. Third, the simplicity of our approach in comparison to the task of formally verifying, e.g. IP and TCP [5], opens up for the possibility of extending currently ongoing work on formally verified low level software along the lines of seL4 [25] to fully networked operating systems and hypervisors at the device and instruction level. Finally, even if amending the current ip naming schemes turns out to be infeasible, it is still of interest to evaluate the consequences of a much tighter integration between network infrastructure and application level runtimes than what is currently done.

Our study is set in the context of a distributed object language $\mu$ABS, a rudimentary fragment of the ABS language studied in the EU FP7 project HATS. The ABS language [22] is an intermediate level modeling and programming language in which a number of phenomena related to software evolvability and adaptation can be studied. The $\mu$ABS fragment includes only a rudimentary set of features, sufficient, however, to allow simple networked programs to be programmed in a natural way, as we illustrate by a couple of small examples. The $\mu$ABS language is class based, and has features for remote method invocation, object creation, and standard sequential control structures, similar to Sewell et al’s Nomadic Pict language [36]. Two semantics of $\mu$ABS are given, one a standard semantics in rewriting logic style which does not take into account aspects related to location, naming, routing, or communication. The second semantics takes these aspects into account by executing objects on top of an arbitrary, but concrete, processor network. The main result of the paper is to show that the network semantics is sound and fully abstract with respect to the reference semantics. We base the analysis on barbed equivalence [33]. Barbed equivalence requires a witness relation that preserves some set of primitive observations, here remote calls to external objects, in both directions, and is preserved under weak reductions, also in both directions. Barbed equivalence is convenient since the required (unlabelled) reduction relations are easy and fairly uncontroversial to define, both for the reference semantics and for the network semantics. Using a labelled transition semantics particularly at the network level is much more complex, and needs to be subject to a separate justification which is out of scope for the present work.

We structure the presentation as follows: First, in sections 3 to 5 the $\mu$ABS language is introduced, along with the reference semantics and the notion of barbed equivalence. We then turn to the network semantics. In section 6 we first introduce runtime states, or configurations, including routing and network graphs. The reduction relation is presented in section 7. The subsequent analysis relies on a collection of simple well-formedness conditions that are introduced in section 8, along with a proof that reachable configurations are well-formed. After adapting barbed equivalence to the network semantics we then, in section 10, embark on the soundness and
full abstraction proof. A key tool is two normal form theorems which allow configurations in the network semantics to be normalized in a behaviour preserving way. The first normal form theorem allows to stabilize routing and at least partially empty message queues. The second normal form theorem serves to as much as possible empty message queues and to migrate all objects to a single central node. Once this is done, soundness and full abstraction is proved, in section 11. The proofs are a bit lengthy but not really complicated, mainly due to the simplicity of the source language. Finally in section 12 we discuss future and related work, including ongoing extensions of this work to richer source languages [11] and to resource allocation [10].

Acknowledgements  Many thanks are due to Karl Palmskog for valuable contributions in the early phases of this work, and to Andreas Lundblad and Ali Jafari.

2 Notation

We use a standard boldface vector notation to abbreviate sequences, for compactness. Thus, \( x \) abbreviates a sequence \( x_0, \ldots, x_n \), possibly empty, and \( f(x) \) abbreviates a sequence \( f_1 x_1, \ldots, f_n x_n \), etc. Let \( x = x_1, \ldots, x_n \). Then \( x_0, x \) abbreviates \( x_0, \ldots, x_n \). Let \( g : A \rightarrow B \) be a finite map. The update operation for \( g \) is \( g[b/a] \) if \( x \neq a \) and \( g[b/a](a) = b \). We use \( \perp \) for bottom elements, and \( A_\perp \) for the lifted set with partial order \( \sqsubseteq \) such that \( a \sqsubseteq b \) if and only if either \( a = b \in A \) or else \( a = \perp \). Also, if \( x \) is variable ranging over \( A \) we often use \( x_\perp \) as a variable ranging over \( A_\perp \). For \( g \) a function \( g : A \rightarrow B_\perp \) we write \( g(a) \downarrow \) if \( g(a) \in B \), and \( g(a) \uparrow \) if \( g(a) = \perp \). The product of sets (flat cpo’s) \( A \) and \( B \) is \( A \times B \) with pairing \( (a, b) \) and projections \( \pi_1 \) and \( \pi_2 \).

3 \( \mu \text{ABS} \)

The syntax of the object language \( \mu \text{ABS} \) is presented in fig. 1. Programs are

<table>
<thead>
<tr>
<th>Variables</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x, y \in \text{Var} )</td>
<td>( e \in \text{Exp} )</td>
</tr>
<tr>
<td>( P ::= CL[x, s] )</td>
<td>Program</td>
</tr>
<tr>
<td>( CL ::= \text{class } C(x){y, M} )</td>
<td>Class definition</td>
</tr>
<tr>
<td>( M ::= m(x){y, s} )</td>
<td>Method definition</td>
</tr>
<tr>
<td>( s ::= s_1; s_2 \mid x = \text{rhs} \mid \text{skip} \mid \text{if } e{s_1} \text{ else } {s_2} \mid \text{while } e{s} \mid e m(e) )</td>
<td>Statement</td>
</tr>
<tr>
<td>( \text{rhs ::= } e \mid \text{new } C(e) )</td>
<td>Right hand sides</td>
</tr>
</tbody>
</table>

Figure 1: \( \mu \text{ABS}_1 \) abstract syntax

sequences of class definitions, along with a set of global variables \( x \), and a "main" statement \( s \). The class hierarchy is flat and fixed. Objects have
class Server1()
{
  serve(from,x)
  {
    from!response(foo(x))
  }
}

class Client(arg)
{
  use(server)
  {
    server!serve(self,arg)
  }
  response(y)
  {
    env!output(y)
  }
}

{
  server, client,
  server = new Server1()
  ;
  client = new Client(42)
  ;
  client!use(server)
}

Figure 2: Simple server

parameters $x$, local variable declarations $y$, and methods $M$. Methods have parameters $x$, local variable declarations $y$ and a statement body. For simplicity we assume that variables have unique declarations. The definition of expressions $e$ is left open, but we require that expressions are side-effect free. Types are omitted from this presentation. It is possible to add types and a notion of well-typedness. However, this will not affect the presentation in any significant way, and for this reason we choose to work in an untyped setting.

Besides standard sequential control structures (the choice of which is largely irrelevant), statements involve a minimal set of constructs for asynchronous method invocation and object creation. Sequential composition is associative with unit skip. That is, the statements $s;skip$, $skip;s$ and $s$ are identified. Method bodies lack a return statement. We consider return statements with futures in a companion paper [11]. For now, method bodies are simply evaluated to the end at which point the evaluating task is terminated. In the absence of return statements, objects communicate using callbacks in a manner which is not dissimilar to communication in Erlang, as illustrated in the following examples.

**Example 3.1.** A very simple server applying $foo$ to its argument is shown in fig. 2. We assume a reserved OID env with reserved method output to be used as a standard output channel.

**Example 3.2.** Just to show that the language is not trivial, the program in fig. 3 constructs an object ring with (here) 42 elements. A value is circulated along the ring, computing $bar(...(bar(x,42),41)...,1)$. Each ring element decrements an iterator $iter$ initialized to the original value 42 first received. The final ring element returns the final value to the server, which then finally returns it to the client through the output method of object env.
```java
1: class Server(){,
2:   serve(from,x){c,
3:     c = new Cell() ;
4:     c!process(x,self,x)}
5:   return(result){from!response(result)}
6: }
7: class Cell(){,
8:   process(x,root,iter){c,
9:     if iter = 0{root!return(y)}
10:    else {
11:      c = new Cell() ;
12:      c!process(bar(x,iter),root,iter -1})
13:  }
14: },
15: class Client(arg){,
16:   use(server){,server!serve(self(),arg)},
17:   response(y){,env!output(y) }
18: }
19: {
20: server, client,
21: server = new Server() ;
22: client = new Client(42) ;
23: client!use(server)
24: }
```

Figure 3: Dynamic ring
4 Type 1 Reduction Semantics

We first present a reduction semantics for $\mu$ABS in rewriting logic style. This semantics is important as the point of reference for later refinements. The reduction semantics uses a reduction relation $cn \rightarrow cn'$ where $cn$, $cn'$ are configurations, as determined by the runtime syntax in fig. 4. Later on, we introduce different configurations and transition relations, and so use index 1, or talk of e.g. configurations of “type 1”, for this first semantics when we need to disambiguate. Terms of the runtime syntax are ranged over by $M$, $x \in Var$ Variables

Variables

Object id

Primitive values

Values

Method environment

Object environment

Task

Object

External call

Container

Configuration

Figure 4: $\mu$ABS$_1$ type 1 runtime syntax

and $\prec$ is the subterm relation. The runtime syntax uses disjoint, denumerable sets of object identifiers $o \in OID$ and primitive values $p \in PVal$. Lifted values are ranged over by $v_\perp \in Val_\perp$. We often refer to OID’s as names, and bind OID’s using the $\pi$-like binder $bind$. The free names of configuration $cn$ is the set $fn(cn)$, as usual, and $OID(cn) = \{o \mid \exists a.o(a, a) \not\approx cn\}$ is the set of OID’s of objects occurring in $cn$. Standard alpha-congruence applies to name binding (but is dropped once we move to the network semantics).

In order for computations to have observable effects we assume a fixed set $Ext$ of external OID’s with dedicated methods, such as the OID $env$ and the method $output$ in examples 2 and 3. External OID’s are not allowed to be bound.

Method and object environments $l$ and $a$, respectively, map local variables to assignable values. Upon invocation, the method environment is initialized using the operation $locals(o, m, v)$ by mapping the formal parameters of $m$ in $o$ to the corresponding actual parameters in $v$, by initializing the method local variables to a suitable null value, and by mapping $self$ to $o$. Object environments are initialized using the operation $init(C, v)$, which maps the parameters of $C$ to $v$, and initializes the object local variables as above.

Configurations are multisets of containers of which there are three types, tasks, objects, and external calls. Object identifiers are scoped within configurations using the $\pi$-like binder $bind$. Configuration juxtaposition is assumed to be commutative and associative with unit 0. In addition we assume the standard structural identities $bind o.0 = 0$ and $bind o.(cn_1 \ cn_2) = (bind o.cn_1) \ cn_2$ when $o \not\in fn(cn_2)$. We often use a vectorized notation

\begin{align*}
\begin{array}{l}
x \in Var \\
o \in OID \\
p \in PVal \\
v \in Val \\
l \in MEnv = Var \rightarrow Val \\
a \in OEnv = Var \rightarrow Val \\
tsk \in Tsk ::= t(o, l, s) \\
obj \in Obj ::= o(o, a) \\
cl \in Call ::= c(o, o', m, v) \\
ct \in Ct ::= tsk | obj \\
cn \in Cn ::= 0 | ct | cn cn' | bind o.cn
\end{array}
\end{align*}
bind o.cn as abbreviation, letting bind ε.cn = cn where ε is the empty sequence. The structural identities then allows us to rewrite each configuration into a standard form bind o.cn such that each member of o occurs free in cn, and cn has no occurrences of the binding operator bind. We use standard forms frequently below.

In addition to locals and init, the reduction rules presented below use the following helper functions:

- body(o, m) retrieves the statement of the shape \( s \) in the definition body for \( m \) in the class of \( o \)
- \( \hat{e}(a, l) \in Val \) evaluates \( e \) using method environment \( l \) and object environment \( o \)

Call containers play a special, somewhat subtle role in defining the external observations of a configuration \( cn \). An observation, or barb, is a call expression of the form \( o!m(v) \), ranged over by \( obs \). In order to define the observations of a given configuration, we assume a fixed set \( Ext \) of external OID’s to which outgoing method calls can be directed. Names in \( Ext \) are not allowed to be bound. A barb, then, is an external method call, i.e. a method call to an OID in \( Ext \). Calls that are not external are meant to be completed in usual reduction semantics style, by internal reaction with the called object, to spawn a new task. External calls could be represented directly, without introducing a special container type (which is not present in the core ABS semantics of [22]), by saying that a configuration \( cn \) has barb \( obs = o!m(v) \) if and only if \( cn \) has the shape

\[
\text{bind } o_1.o(o_2, a) t(o_2, l, e_1!m(e_2); s),
\]

where \( e_1(a, l) = o \in Ext \) and \( e_2(a, l) = v \). However, in a semantics with unordered communication, which is what is assumed of core ABS [22], and which we also implement in this paper; consecutive calls should commute, i.e. there should be no observational distinction between the method bodies \( e_1!m_1(e'_1); e_2!m_2(e'_2) \) and \( e_2!m_2(e'_2); e_1!m_1(e'_1) \). This, however, is difficult to reconcile with the representation (1). To this end call containers are introduced, to allow configurations like (1) to produce a corresponding call, and then proceed to elaborate \( s \).

Figures 5 and 6 present the reduction rules, using the notation \( cn \vdash cn' \rightarrow cn'' \) as shorthand for \( cn \rightarrow cn' \rightarrow cn'' \). We use \( \rightarrow_1 \) when we want to make the reference to the type 1 reduction semantics explicit. Fig. 5 gives the mostly routine rules for assignment, control structures, and contextual reasoning, and fig. 6 gives the slightly more interesting rules for inter-method communication and object creation. We note some basic properties of the reduction semantics.

**Proposition 4.1.** Let \( cn \rightarrow_1 cn' \).

1. \( fn(cn) \subseteq fn(cn') \)
2. If \( o(a, a') \preceq cn \) then \( o(a, a') \preceq cn' \) for some object environment \( a' \)
ctx-1: If $cn_1 \rightarrow cn_2$ then $cn \vdash cn_1 \rightarrow cn_2$

ctx-2: If $cn_1 \rightarrow cn_2$ then bind $o.cn_1 \rightarrow$ bind $o.cn_2$

wlocal: If $x \in dom(l)$ then $t(o, l, x = e; s) \rightarrow t(o, l[\hat{e}(a, l)/x], s)$

wfield: If $x \in dom(a)$ then $o(a) t(o, l, x = e; s) \rightarrow o(a, a[\hat{e}(a, l)/x]) t(o, l, s)$

skip: $t(o, l, \text{skip}) \rightarrow 0$

if-true: If $\hat{e}(a, l) \neq 0$ then $o(a) \vdash t(o, l, \text{if } e\{s_1\} \text{ else } s_2; s) \rightarrow t(o, l, s_1; s)$

if-false: If $\hat{e}(a, l) = 0$ then $o(a) \vdash t(o, l, \text{if } e\{s_1\} \text{ else } s_2; s) \rightarrow t(o, l, s_2; s)$

while-true: If $\hat{e}(a, l) \neq 0$ then $o(a) \vdash t(o, l, \text{while } e\{s_1\}; s) \rightarrow t(o, l, s_1; \text{while } e\{s_1\}; s)$

while-false: If $\hat{e}(a, l) = 0$ then $o(a) \vdash t(o, l, \text{while } e\{s_1\}; s) \rightarrow t(o, l, s)$

Figure 6: µABS reduction rules part 2

Proof. We note that no structural identity nor any reduction rule allows an OID to escape its binder. The result follows. \qed

Consider a program $CL[x, s]$. Assume a reserved OID main. A type 1 initial configuration is any configuration of the shape

$cn_{init} = o(main, \bot) t(main, l_{init}, s)$

where $\bot$ is the everywhere undefined object environment and $l_{init}$ is the initial type 1 method environment assigning default values to the variables in x.

We say that a configuration $cn_n$ of type 1 is reachable if there is a derivation $cn_{init} = cn_0 \rightarrow_1 \cdots \rightarrow_1 cn_n$ where $cn_{init}$ is an initial configuration. Reachable configurations satisfy some well-formedness conditions which we note.

**Definition 4.2** (Type 1 Well-formedness). A configuration $cn$ is type 1 well-formed (WF1) if $cn$ satisfies:

1. OID Uniqueness: Suppose $o(a_1, a_1), o(a_2, a_2) \preceq cn$ are distinct object occurrences. Then $a_1 \neq a_2$

2. Task-Object Existence: If $t(o, l, s) \preceq cn$ then $o(a, a) \preceq cn$ for some object environment $a$

---

call: Let $o' = \hat{c}_1(a, l)$ in $o(a, o) o(o', a') \vdash t(o, l, e_1!m(e_2); s) \rightarrow t(o, l, s) t(o', \text{locals}(o', m, e_2(a, l)), \text{body}(o', m))$

call-ext: If $o' = \hat{c}_1(a, l) \in \text{Ext}$ then $o(a, o) \vdash t(o, l, e_1!m(e_2); s) \rightarrow t(o, l, s) c(a, o', m, e_2(a, l))$

new: $o(a, o) \vdash t(o, l, x = \text{new } C(e); s) \rightarrow \text{bind } o'.t(o, l[o'/x], s) o(a', \text{init}(C, \hat{e}(a, l)))$

---

Figure 6: µABS reduction rules part 2
3. Object Existence: Suppose $o \not\in \text{Ext}$ occurs in $cn$. Then $o(a, o) \preceq cn$ for some object environment $a$

4. Object Nonexistence: Suppose $o \in \text{Ext}$. Then $o(a, o) \not\preceq cn$ for any object environment $a$

5. Object Binding: Suppose $o \not\in \text{Ext}$. Then $o \not\in fn(cn)$

Well-formedness is important as it ensures that objects, if defined, are defined uniquely. The existence properties are important to make sure that the partitioning of OID’s into external and (by extension) internal is meaningful, in that external references are always routed outside the “current configuration”.

**Proposition 4.3 (WF1 Preservation).** If $cn$ is type 1 wellformed and $cn \rightarrow cn'$ then $cn'$ is type 1 wellformed.

**Proof.** Routine. \qed

**Theorem 4.4.** If $cn$ is type 1 reachable then $cn$ is fully type 1 wellformed.

**Proof.** It suffices to check that any initial configuration is fully type 1 well-formed, and then use proposition 4.3. \qed

## 5 Barbed Equivalence

Our approach to implementation correctness is based on the notion of barbed equivalence [33], a notion of equivalence often used to relate transition systems determined by a reduction semantics, cf. [8, 15, 18]. Our goal is to show that it is possible to remain strongly faithful to the reference semantics, provided all nondeterminism is deferred to be handled by a separate scheduler. This allows to draw strong conclusions also in the case a scheduler is added, as we discuss later. Barbed equivalence requires of a pair of equivalent configurations that the internal transition relation $\rightarrow$ is preserved in both directions, while preserving also a set of external observations. Although weaker than corresponding equivalences such as bisimulation equivalence on labelled transition systems, barbed equivalence in nonetheless of interest for the following two reasons:

1. Barbed equivalence offers a reasonable account of observationally identical behaviour on closed systems, i.e. when composition of (in our case) subconfigurations to build larger configurations is not considered because it a) is for some reason not important or relevant, or b) does not offer new observational capabilities.

2. Barbed equivalence can be strengthened in a natural way to contextual equivalence [32] by adding to barbed equivalence a natural requirement of closure under context composition. Furthermore, a number of works [21, 34] have established very strong relations between contextual equivalence for reduction oriented semantics and bisimulation/logical relation based equivalences for sequential and higher-order computational models.
It is, however, far from trivial to devise a natural notion of context that works at the level of the network semantics introduced later, and such that the notions of context correspond at both the abstract, reference semantics level we consider at present, and at the network level. For this reason the account of this paper based on barbed equivalence is also a natural stepping stone towards a deeper study of the notion of context in real-world—or at least not overly artificial—networked software systems.

Let $obs = o'!m(v)$. The observation predicate $cn \downarrow obs$ is defined to hold just in case $cn$ can be written in the form

$$bind \, o.(cn', c(o', m, v)).$$

The derived predicate $cn \Downarrow obs$ holds just in case $cn \Rightarrow^* cn' \downarrow obs$ for some $cn'$.

Let now $R$ be a binary relation on type 1 well-formed configurations. We are interested in relations with the following properties:

- **Symmetry**: If $cn_1 R cn_2$ then $cn_2 R cn_1$
- **Reduction-closure**: If $cn_1 R cn_2$ and $cn_1 \Rightarrow cn'_1$ then there exists some $cn'_2$ such that $cn_2 \Rightarrow^* cn'_2$ and $cn'_1 R cn'_2$
- **Barb preservation**: If $cn_1 R cn_2$ and $cn_1 \downarrow obs$ then $cn_2 \Downarrow obs$

We call a relation with these three properties a **type 1 witness relation**.

**Definition 5.1 (Type 1 Barbed Equivalence).** Let $cn_1 \simeq_1 cn_2$ if, and only if, $cn_1 R cn_2$ for some type 1 witness relation $R$.

Barbed equivalence is the reference behavioral identity to which other equivalences are compared in the remainder of the paper.

**Example 5.2.** Even if this is somewhat out of scope we sketch for completeness a proof that the programs of example 2 and 3 are barbed equivalent. The external OID’s is the single env. We construct a relation $R$ relating configurations of the simple server with configurations of the dynamic ring. For the former, a reachable configuration has one of the following three forms:

- The initial configuration with an object with OID main and a task at line 10
- The initial object, a task at l. 11, a Server object
- The initial object, a task at l. 12, a Server object, and a Client object
- The initial object, a Server object, a Client object, and an invoked use task
- The initial object, a Server object, a Client object, and an invoked serve task
- The initial object, a Server object, a Client object, and an invoked response task
All configurations have exactly one observation, namely env!output(42). For the dynamic ring, the state space is a little more complex. It contains:

- The initial configuration with an object with OID main and a task at line 20
- The initial object, a task at l. 21, a Server object
- The initial object, a task at l. 22, a Server object, and a Client object
- The initial object, a Server object, a Client object, and an invoked use task
- The initial object, a Server object, a Client object, and an invoked serve task
- The initial object, a Server object, a Client object, 42 Cell objects, and a process task at one of lines 9-12, invoked with arguments bar(...(bar(42,42),41)...,42−n), OID of Server, and iter = 42.
- The initial object, a Server object, a Client object, 42 Cell objects, and a response task ready to execute env!output(42).

It is straightforward though somewhat cumbersome to verify that the relation obtained by relating any of the reachable pairs \((cn_1, cn_2)\) where \(cn_1\) is a configuration of the simple server and \(cn_2\) is a configuration of the dynamic ring is a barbed bisimulation.

An important shortcoming of barbed equivalence as we have introduced it here is that prima facie it does not take program structure, or configuration structure, into account. Thus, there is no guarantee, for instance, that if \(cn_1 \simeq_1 cn_2\) then, for all \(cn, cn_1 cn \simeq_1 cn_2 cn\). With the current static internal/external partitioning of OID’s the question is not that meaningful, as e.g. \(cn\) and \(cn’\) are unable to communicate when both are closed. But closedness is important to equip configurations with clearly defined interfaces. It is possible to refine the notion of wellformedness to better capture composition of configurations. This allows to consider a form of contextual equivalence [32], by extending the three conditions above with a condition of contextuality, saying that whenever \(cn_1 \mathcal{R} cn_2\) (subject to suitable wellformedness constraints) then \(cn_1 cn \mathcal{R} cn_2 cn\). This, however, complicates matters quite considerably when we turn to the network semantics below, and for this reason we defer a proper treatment of network composition and contextual equivalence to a future publication.

6 Network Semantics: Runtime Configurations

The “standard” (type 1) semantics for \(\mu\text{ABS}\) is quite abstract and does not account for many issues which must be faced by an actual implementation, in particular if the goal is high performance and scalability. For instance:
• The µABS semantics implements a rendez-vous oriented communication model. We want to account for this using a standard buffered asynchronous model

• Accordingly, calls should be replaced by message passing

• The µABS semantics has no concept of proximity or name space. Any two objects, regardless of their ”location” can without any overhead or search choose to synchronize at any point. Instead, we want a semantics that is network aware in the sense that it brings out proximity and location without unduly constraining the model, for instance to a particular naming discipline, or to a centralized name or location lookup service

Our proposal is to execute µABS objects on a network graph in a fully decentralized and lock free manner where the only means of communication or synchronization is by asynchronous message passing along edges connecting neighbouring nodes, each edge having an associated directional, buffered communication channel. In this section we accordingly introduce a refinement of the standard semantics, a “network semantics”, or type 2 semantics, which adds an explicit network components to the type 1 semantics. The key idea is to use name-based routing, as explained in the introduction. That is, nodes are equipped with explicit routing information allowing messages to be addressed to specific receiving objects, rather than their hosts, which may change. This allows a very simple, fully decentralized, and lock free integration of routing and object migration, as we now begin to demonstrate.

Nodes and Routing  The network semantics is presented in rewriting logic style, similar to the type 1 semantics above. We still have configurations $cn$, but these now have a richer structure. We first introduce two new types of container to reflect the underlying network graph, namely nodes and links. Node containers have the form

$$n(u, t)$$

where $u \in NID$ is a primitive node identifier, and where $t$ is an associated routing table. Node identifiers (NID’s) take the place of ip addresses in the usual ip infrastructure. For routing we assume a rudimentary Bellman-Ford distance vector (d.v.) routing discipline [39]. More elaborate and practical routing schemes exist that are better equipped for e.g. disconnected operation, and with better combinations of scalability and stretch. However, for the present purpose, the d.v. scheme achieves its purpose. Consequently, a routing table $t$ is a partial function associating to the OID’s $o$ ”known” to $t$ a pair $t(o) = (u, n)$ where $n$ is the minimum number of hops believed by $t$ to be needed to reach the node hosting $o$ from the current node, and where $u$ is the next hop destination.
Routing Tables  Routing tables support the following operations:

- Next hop lookup, \(\text{nxt}(o, t) = \pi_1(t(o))\): In the context of a node \(n(u, t)\), \(\text{nxt}(o, t)\) returns a neighbour \(u'\) of \(u\) to which, according to the current state of \(u\), a message should be sent in order to eventually reach the destination \(o\).

- Update, \(\text{upd}(t, u, t')\): Updates \(t\) by incorporating the routing table \(t'\) belonging to a (neighbouring) node \(u\). The update function is defined thus:

\[
\text{upd}(t, u, t')(o) = \begin{cases} 
\bot & \text{if } o \notin \text{dom}(t) \cup \text{dom}(t') \\
\text{t}(o) & \text{else, if } o \notin \text{dom}(t) \\
(u, \pi_2(t'(o)) + 1) & \text{else, if } o \notin \text{dom}(t) \\
(u, \pi_2(t'(o)) + 1) & \text{else, if } \pi_1(t'(o)) = u \\
(u, \pi_2(t'(o)) + 1) & \text{else, if } t'(o) < \pi_2(t(o)) - 1 \\
\text{t}(o) & \text{otherwise}
\end{cases}
\]

If \(o\) is known to neither the current node nor to \(u\), the distance estimate to \(o\) from the current node is undefined. If it is known to the current node, but not to \(u\), \(t\)'s information is unchanged. If it is known to \(u\), but not to the current node, the estimate from the current node becomes \(1 + u\)'s estimate. Otherwise we may assume that \(u\) is known to both the current node and to \(u\). If the minimal route follows the edge between the current node and \(u\), \(u\)'s distance estimate plus one is the new distance estimate at the current node, regardless of whether the estimate is improves on the current estimate or not. Otherwise, if \(u\)'s estimate improves sufficiently on the estimate at the current node, \(u\)'s estimate is incremented and used at the current node. In other circumstances, the distance estimate at the current node is left unchanged.

- Registration, \(\text{reg}(o, u, t)\): Returns the routing table \(t'\) obtained by registering \(o\) at \(u\) in \(t\), i.e.

\[
\text{reg}(o, u, t)(o') = \begin{cases} 
(u, 0) & \text{if } o = o' \\
\text{t}(o') & \text{otherwise}
\end{cases}
\]

The function \(\text{reg}\) is invoked only when \(u\) is the “current” node.

Links, Queues, and Messages  Nodes are connected by directed edges, or links, of the form

\(l(u, q, u')\),

where \(u \in \text{NID}\) is the source NID, \(u' \in \text{NID}\) is the sink NID, and where \(q \in Q\) is the associated fifo message queue. Queue operations are standard: \(\text{enq}(msg, q)\) enqueues the message \(msg\) onto the tail of \(q\), \(\text{hd}(q)\) returns the head of \(q\), and \(\text{deq}(q)\) returns the tail of the \(q\), i.e. \(q\) with \(\text{hd}(q)\) removed. If \(q\) is empty (\(q = \epsilon\)) then \(\text{hd}(q)\) and \(\text{deq}(q)\) are both undefined.

Messages have one of the following three forms:
call(o,o′,m,v): A remote call message originating from object o and addressed to object o′, of method m, and with arguments v.

table(t): A routing table update message. The origin NID is implicit, as the message is dequeued from a link queue with explicit source NID.

object(cn): An object migration message, where cn is an object closure, as explained below.

Call messages are said to be object bound, and table and object messages are said to be node bound. We define dst(msg), the destination of msg to be o′ for call messages, and dst(msg) = ⊥ in the remaining two cases.

The Network Graph  Nodes and links induce a directed graph structure graph(cn) in the obvious way, by taking as vertices the NID’s u and as edges pairs (u,u′) for each link l(u,q,u′). For this to make sense we impose some constraints that apply from now on, to all “global” configurations cn in the type 2 semantics.

1. Unique vertices: There is at most one container n(u′,t) ∈ cn with u′ = u

2. Unique edges: For each source-sink pair u,u′ there is at most one link l(u,q,u′), for some q

3. Edges connect vertices: If l(u,q,u′) ∈ cn then n(u,t),n(u′,t′) ∈ cn for some t,t′

4. Reflexivity: graph(cn) is reflexive, i.e. if n(u,t) ∈ cn for some t then l(u,q,u) ∈ cn for some q

5. Symmetry: graph(cn) is symmetric, i.e. if l(u,q,u′) ∈ cn then l(u′,q′,u) ∈ cn for some q′

6. Connectedness: graph(cn) is connected, i.e. if n(u,t),n(u′,t′) ∈ cn then there is a path in graph(cn) connecting vertices u and u′

Conditions 1 is essential for naming. Condition 2 is important in the present paper, as the present paper focuses on closed systems. Condition 3 simplifies communication but could be lifted in principle. Conditions 4 and 5 are non-essential, but helpful. Finally, condition 6 is essential for routing to stabilize, but many of the results below can be proved without it.

Objects and Tasks  In the type 2 semantics object containers are now attached to a node u and have the shape

\[ o(a,a,u,q_{in},q_{out}) \]

where o ∈ OID, a ∈ OEnv as before, and q_{in}, q_{out} is a pair of an ingoing and an outgoing fifo message queue. This object level buffering is not essential, as messages are already buffered at link level, but object level buffering
allows a more elegant formalization. It is commonplace in actor languages to consider inbound queues only. Here we find it more elegant to allow an outgoing queue as well, although this is mainly a matter of taste. Tasks \( t(o, l, s) \) are unchanged from the type 1 semantics.

**Object Closures**  Type 2 configurations are built from the four container types introduced above, nodes, links, objects and tasks. It remains to explain object closures. For an object message object \((cn)\) to be valid, the configuration \(cn\) needs to be an object closure of the form

\[
o(o, a, u, q_{in}, q_{out}) t(o, l_1, s_1) \ldots t(o, l_n, s_n)
\]

Specifically, if \(cn\) is any configuration then \(clo(cn, o)\), the closure of object \(o\) with respect to \(cn\), is the multiset of all type 2 containers of the form either \(o(o', a', u', q'_{in}, q'_{out})\) or \(t(o', l', s')\) such that \(o' = o\), and \(objf(cn)\) is a partial function returning \(o\) if all type 2 containers in \(cn\) are either objects or tasks, with OID \(o\).

**Type 2 Runtime Syntax**  Reflecting the above description, the type 2 runtime syntax is presented in fig. 7. A pictorial representation of the type 2 runtime state is shown in fig. 8. As implicit above, configurations remain multisets, and we write, e.g., \(obj \in cn\) if \(cn\) can be written as \(obj \ cn'\) for some \(cn'\). Tasks are unchanged from fig. 4. We write \(t(cn)\) for the multiset of tasks in \(cn\), i.e. the multiset \(\{tsk \mid \exists cn', cn = tsk \ cn'\}\), and \(o(cn)\) for the multiset of objects in \(cn\), similarly defined. We also write \(m(cn)\) for the multiset \(\{msg \mid msg \preceq cn\}\). To avoid explosion of the notation we reuse symbols from the type 1 semantics as far as possible, and resolve them by context.

### Figure 7: Type 2 runtime syntax

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(u)</td>
<td>(NID)</td>
</tr>
<tr>
<td>(t)</td>
<td>(RTable) = (OID \rightarrow NID \times \omega)</td>
</tr>
<tr>
<td>(q)</td>
<td>(Q) = (Msg^*)</td>
</tr>
<tr>
<td>(obj)</td>
<td>(Obj_2) ::= (o(o, a, u, q_{in}, q_{out}))</td>
</tr>
<tr>
<td>(nd)</td>
<td>(Nd) ::= (n(u, t))</td>
</tr>
<tr>
<td>(lnk)</td>
<td>(Lnk) ::= (l(u, q, u'))</td>
</tr>
<tr>
<td>(ct)</td>
<td>(Ct_2) ::= (tsk \mid obj \mid nd \mid lnk)</td>
</tr>
<tr>
<td>(cn)</td>
<td>(Cn_2) ::= (ct_1 \ldots ct_n)</td>
</tr>
<tr>
<td>(msg)</td>
<td>(Msg) ::= (call(o, o', f, m, v) \mid table(t) \mid object(cn))</td>
</tr>
</tbody>
</table>

7 **Type 2 Reductions**

An important distinction between the standard semantics and the network semantics is the absence of binding. For the standard semantics, name binding plays an key role to avoid clashes between locally generated names.
However, in a language with NID’s this device is no longer needed, as globally unique name can be guaranteed easily by augmenting names with their generating NID. Since all name generation takes place in the context of a given NID, we can simply assume an operation newO(u) that return a new OID, which is globally fresh for the “current configuration”. Another important point to note is that all transitions in the type 2 are fully local, in the sense that all operations applied, and all conditions determining whether or not a transition is enabled, can be fully determined by inspecting only one node and, possibly, the head of incoming link queues, alternatively by enqueuing messages to the tail of the outgoing queue.

First, the rules in fig. 5 apply with the following two minor modifications:

- Rule ctxt-2 is dropped as name binding is dropped from the type 2 runtime syntax
- Rule wfield is modified in the obvious way to read: If $x \in \text{dom}(a)$ then $o(a, a, u, q_{in}, q_{out}) \xrightarrow{t(o, l, x = e; s)} o(a, a[e(a, l)/x], u, q_{in}, q_{out}) \xrightarrow{t(o, l, s)}$

The remaining reduction rules are presented in fig. 9. The rules are naturally divided into subgroups:

- The rules t-send and t-rcv are concerned with the exchange of routing tables
- The three rules msg-send, msg-rcv and msg-route are used to manage message passing, i.e. reading a message from a link queue and transferring it to the appropriate object in-queue, and dually, reading a message from an out-queue and transferring it to the attached link queue. Finally, messages are routed to the next link, if the destination object does not reside at the current node. In rule msg-rcv note that the receiving node is not required to be present. This, however, will be enforced by the well-formedness condition later.
t-send: \( n(u, t) \vdash l(u, q, u') \rightarrow l(u, \text{enq}(\text{table}(t), q), u') \)

t-rcv: If \( hd(q) = \text{table}(t') \) then \( l(u', q, u) \vdash l(u', \text{deq}(q), u) \vdash n(u, \text{upd}(t, u', t')) \)

msg-send: If \( hd(q) = msg \), \( dst(msg) = o' \) and \( nxt(o', t) = u' \) then
\[ n(u, t) \vdash l(u, q, u') \circ(o, a, u, qin, qout) \rightarrow l(u, \text{enq}(msg, q), u') \circ(o, a, u, qn, \text{deq}(qout)) \]

msg-rcv: If \( hd(q) = msg \) and \( dst(msg) = o \) then
\[ l(u', q, u) \circ(o, a, u, qin, qout) \rightarrow l(u', \text{deq}(q), u) \circ(o, a, u, \text{enq}(msg, q), q_{out}) \]

msg-route: If \( hd(q) = msg \), \( dst(msg) = o \) and \( nxt(o, t) = u'' \neq u \) then
\[ n(u, t) \vdash l(u', q, u) l(u, q', u) \rightarrow l(u', \text{deq}(q), u) l(u, \text{enq}(msg, q), u'') \]

msg-delay-1: If \( hd(q) = msg \), \( dst(msg) = o \) and \( nxt(o, t) \uparrow \) then
\[ n(u, t) \vdash l(u', q, u) l(u, q', u) \rightarrow l(u', \text{deq}(q), u) l(u, \text{enq}(msg, q), u) \]

msg-delay-2: If \( hd(q_{out}) = msg \), \( dst(msg) = o' \), and \( nxt(o', t) \uparrow \) then
\[ n(u, t) \vdash o(o, a, u, q_{in}, q_{out}) l(u, q, u) \rightarrow o(o, a, u, q_{in}, \text{deq}(q_{out})) l(u, \text{enq}(msg, q), u) \]

call-send: Let \( o' = e1(a, l) \in \circ(o, a, u, q_{in}, q_{out}) t(o, l, e1!m(e2): s) \rightarrow \)
\[ o(o, a, u, q_{in}, \text{enq}(\circ(o', m, v), q_{out})) t(o, l, s) \]

call-rcv: If \( hd(q_{in}) = \text{call}(o', a, m, v) \) then
\[ o(o, a, u, q_{in}, q_{out}) \rightarrow o(o, a, u, \text{deq}(q_{in}), q_{out}) t(o, \text{locals}(a, m, v), \text{body}(a, m)) \]

call-recv: Let \( cn' = \text{new}(c) \in \circ(o, a, u, q_{in}, q_{out}) \vdash n(u, t) t(o, l, x = \text{new} C(\vec{e}); s) \rightarrow \)
\[ n(u, \text{reg}(o', u, t)) t(o, l[o'/x], s) o(o', \text{init}(C, \vec{e}(a, l)), u, x, \vec{e}) \]

obj-send: Let \( cn' = \text{clo}(cn, o) \in \circ(o, a, u, q_{in}, q_{out}) \vdash n(u, \text{reg}(o, u', t)) l(u, \text{enq}(\text{object}(cn'), q), u') \) \( cn - cn' \)

obj-recv: If \( hd(q) = \text{object}(cn') \) then
\[ l(u', q, u) n(u, t) \rightarrow l(u', \text{deq}(q), u) n(u, \text{reg}(\text{obj}(cn'), u, t)) cn' \]

---

**Figure 9: Type 2 reduction rules**

- The two rules msg-delay-1 and msg-delay-2 are used to handle the case where routing tables have not yet stabilized. For instance it may happen that updates to the routing tables have not yet caught up with object migration. In this case, a message may enter an out-queue without the hosting nodes routing table having information about the message’s destination (rule msg-delay-2). Another case is where a node receives a message on a link without knowing where to forward it (rule msg-delay-1). This situation is particularly problematic as a blocked message may prevent routing table updates to reach the hosting node, thus causing deadlock. The solution we propose is to use the network self-loop as a buffer for temporarily unroutable messages.

- The rules call-send and call-rcv produce and consume call messages in a pretty obvious way.

- The rule new-2 handles object creation, including registration of the new object at the local node.

---

19
The final two rules concern object migration. Of these, obj-send is a global rule in that it is not allowed to be used in subsequent applications of the ctxt-1 rule. In this way we can guarantee that only complete object closures are migrated. In rule obj-send, \( cn - cn' \) is multiset difference.

We emphasize again that all of the above rules are strictly local and appeal only to mechanisms directly implementable at link level: Tests and simple datatype manipulations taking place at a single node, or accesses to a single nodes link layer interface. The “global” property appealed to above for the migration rules is merely a formal device to enable an elegant treatment of object closures.

The reduction rules can be optimized in several ways. For instance, object self-calls are always routed through the “network interface”, i.e. the hosting nodes self-loop. This is not necessary. It would be possible to add a rule to directly spawn a handling task from a self call without affecting the results of the paper.

We note some elementary properties of the type 2 semantics.

**Proposition 7.1.** Suppose that \( cn \to cn' \).

1. If \( n(u, t) \preceq cn \) then \( n(u, t') \preceq cn' \) for some \( t' \)

2. If \( lnk = l(u, q, u') \preceq cn \) then \( l(u, q', u') \preceq cn' \) for some \( q' \)

3. If \( obj = o(o, u, q_{in}, q_{out}) \preceq cn \) then there is an object

\[
obj' = o(o', u', q_{in}', q_{out}') \preceq cn'
\]

(the derivative of \( obj \) in \( cn' \)) such that \( o' = o, u' = u \), and for all \( x \), if \( a(x) \downarrow \) then \( a'(x) \downarrow \).

**Proof.** By inspecting the rules. \( \square \)

We then turn to initial configurations. Let a program \( CL\{x, s\} \) be given.

**Definition 7.2** (Type 2 Initial Configuration). A type 2 initial configuration has the shape

\[
cn_{init} = cn_{graph} o(main, \perp, u_{init}, \varepsilon, \varepsilon) t(main, l_{init}, s)
\]

where

- \( o(main, \perp) t(main, l_{init}, s) \) is a type 1 initial configuration,

- \( cn_{graph} \) is a configuration consisting only of nodes and links, with empty link queues,

- \( u_{init} \) names a node \( n(u_{init}, l_{init}) \) in \( cn_{graph} \),

- \( l_{init}(main) = (u_{init}, 0) \), and \( l_{init}(o) = \perp \) for \( o \neq main \), and

- \( t(o) = \perp \) for all routing tables \( t \neq l_{init} \) and OID’s \( o \) in \( cn_{init} \).
8 Well-formedness

The well-formedness conditions need to be augmented somewhat for the type 2 semantics. We can observe first that the graph well-formedness conditions of the previous section are clearly preserved by all transitions.

Definition 8.1 (Type 2 Wellformedness). A type 2 configuration $cn$ is type 2 wellformed (WF2) if $cn$ satisfies:

1. **OID uniqueness**: Suppose $o(o_1, a_1, u_1, q_{in, 1}, q_{out, 1}), o(o_2, a_2, u_2, q_{in, 2}, q_{out, 2}) \preceq cn$ are distinct object occurrences. Then $o_1 \neq o_2$

2. **Object-Node existence**: If $o(o, a, u, q_{in}, q_{out}) \in cn$ then $n(u, t) \in cn$ for some $t$.

3. **Task-Object Existence**: If $t(o, l, s) \preceq cn$ then $o(o, a, u, q_{in}, q_{out}) \preceq cn$ for some $a, u, q_{in}, q_{out}$

4. **Object Existence**: Suppose $o \notin Ext$ occurs in $cn$. Then $o(o, a, u, q_{in}, q_{out}) \preceq cn$ for some $a, u, q_{in}, q_{out}$

5. **Object Nonexistence**: Suppose $o \in Ext$. The $o(o, a, u, q_{in}, q_{out}) \not\preceq cn$ for any $a, u, q_{in}, q_{out}$

6. **Buffer cleanliness**: If $o(o, a, u, q_{in}, q_{out}) \preceq cn$ and $msg \preceq q_{in}$ or $msg \preceq q_{out}$ then $msg$ is object bound. Moreover, if $msg \preceq q_{in}$ then $dst(msg) = o$

7. **Local Routing Consistency, 1**: If $n(u, t), o(o, a, u, q_{in}, q_{out}) \in cn$ then $nxt(o, t) = (u, 0)$

8. **Local Routing Consistency, 2**: If $n(u, t) \preceq cn$ and $\pi_1(nxt(o, t)) = u'$ then there is a link $l(u, q, u') \preceq cn$

Most conditions are straightforward. For 8.1.6 observe that only object bound messages (for in-queues, with messages appropriately addressed) enter the object queues. Buffer cleanliness is needed to prevent the formation of contexts that are deadlocked because an in- or out-queue contains messages of the wrong type. For 8.1.7 the requirement should hold only when the object is not in transit (the object is in transit when it is contained in a message queue, i.e. when $\preceq$ holds, but not $\in$), as otherwise the object may be on the wire away from node $u$, and $u$’s routing table will then have been updated. This is not a concern in 8.1.8 as nodes and links never move. WF2 Preservation is very easily verified:

**Lemma 8.2** (WF2 Preservation). If $cn$ is type 2 well-formed and $cn \rightarrow_2 cn'$ then $cn'$ is type 2 wellformed.

As above we note that then all type 2 reachable states are well-formed.

**Corollary 8.3.** If $cn$ is type 2 reachable then $cn$ is fully type 2 wellformed.

**Proof.** First check that initial configurations are type 2 wellformed and closed and then use lemma 8.2.
9 Type 2 Barbed Equivalence

We next adapt the notion of barbed equivalence to the type 2 setting. The only difficulty is to define the type 2 correlate of the observation predicate. We take the point of view that an observation \( \text{obs} = \text{ocl}(v) \) is enabled at a configuration \( cn \) if a corresponding call message \( \text{call}(o',o,m,v) \) is located at the head of one of the object output queues in \( cn \). More precisely, the type 2 observability predicate is \( cn \downarrow \text{obs} \), holding if and only if \( cn \) has the following shape;

\[
\text{cn} = \text{cn'} o(o',a,u,q_{in},q_{out})
\]

and \( \text{hd}(q_{out}) \) is defined and equal to \( \text{call}(o',o,m,v) \).

There are other ways of defining the observability predicate that may be more natural. For instance one may attach external OID’s to specific NID’s and restrict observations to those NID’s accordingly. It is also possible to add dedicated output channels to the model, and route external calls to those. None of these design choices have any effect on the subsequent results, however, but add significant notational overhead, particular in the latter case.

With the observation predicate set up, the weak observation predicate is derived as in section 5, and, as there, we define a type 2 witness relation as a relation that satisfies symmetry, reduction closure, and barb preservation. Thus:

**Definition 9.1.** Type 2 Barbed Equivalence Let \( cn_1 \simeq_2 cn_2 \) if and only if \( cn_1 \mathcal{R} cn_2 \) for some type 2 witness relation \( \mathcal{R} \).

In fact, for the purpose of this paper there in no real need to distinguish between the type 1 and type 2 equivalences, and hence we conflate the notions of witness relation and barbed equivalences, by letting the type of the configuration arguments be determined by the context, and use \( \simeq \) as the generic notion.

10 Normal Forms

The goal is prove that if \( cn_1 \) and \( cn_2 \) are initial type 1 and type 2 configurations, respectively, for the same program, then \( cn_1 \simeq cn_2 \). The key to the proof is a normal form lemma for the type 2 semantics saying, roughly, that any well-formed type 2 configuration can be rewritten, using a subset of the rules as detailed below, into a form where queues have been emptied of all routable messages, where routing tables have been in some expected sense normalized, and where all objects have been moved to a single node. We prove this in two steps. First we prove a stabilization result, that non-self links can be emptied of messages and routing tables normalized to induce messaging paths with unit stretch. This allows the second normalization step to empty also object queues and migrate all objects to a single node. Once this is done we can prove correctness by exhibit a map representing
each type 1 configuration as a canonical type 2 configuration, using normalization to help prove reduction preservation in both direction. Then only barb preservation is needed to complete the correctness argument.

10.1 Stabilization

We first show that each configuration can be rewritten using the transition rules into a form for which routing is stable and all queues are empty, except for external messages, i.e. messages \( \text{msg} \) addressed to an object \( o \in \text{Ext} \). By well-formedness we then know that no object \( o'(a',u',q_{in},q_{out}) \preceq cn \) with \( o' = o \) exists. In the context of a configuration \( cn \) call a proper link any link \( l(u,q,u') \) for which \( u \neq u' \).

**Definition 10.1** (Stable Routing, External Queued Messages). Let \( cn \) be a well-formed type 2 configuration.

1. \( cn \) has **stable routing**, if for all \( n(u,t) \), \( o(a,u',q_{in},q_{out}) \preceq cn \), if \( \text{next}(o,t) = u'' \) then there is a minimal length path from \( u \) to \( u' \) which visits \( u'' \).

2. \( cn \) has **external link messages only**, if \( l(u,q,u') \in cn \) and \( \text{msg} \preceq q \) implies \( u = u' \) and \( \text{msg} \) is external.

The strategy for performing the rewriting is to first empty link queues as far as possible as we simultaneously exchange routing tables to converge to a configuration with stable routing. This first stage is accomplished using algorithm 1 in fig. 10 where we hide uses of ctxt-1 to allow the transition rules to be applied to arbitrary containers. Observe that we have no intention to use alg. 1 or any of the later algorithms in this section to do actual computing in the type 2 semantics. “Real” network computing using the type 2 semantics requires more sophisticated approaches. The algorithms considered here do not need to be effective or “local”: We only need to exhibit some strategy for producing a configuration with the desired result, allowing us to prove the desired normal form results.

**Proposition 10.2.** Algorithm 1 terminates.

**Proof.** See appendix A.

Write \( A_1(cn) \rightarrow cn' \) if the configuration \( cn' \) is a possible result of applying algorithm 1 to \( cn \). We then say that \( cn' \) is in **stable form**. Stable forms are almost unique, but not quite, since routing may stabilize in different ways, and since this (plus the generally nondeterministic scheduling of rules in alg. 1) may cause messages to enter object input queues at different times. Let \( t_1(cn) = \{ \text{tsk} \mid \text{tsk} \preceq cn \} \) and let \( o_1(cn) \) be the multiset of object containers \( ct = o(a,u,q_{in},q_{out}) \) in \( cn \) such that either \( ct \in o_1(cn) \), or else \( o(a,u',q_{in},q_{out}) \) is in transit in \( cn \) from some \( u' \) to \( u \) (since then, after applying alg. 1, \( u \) will host the object). Finally, let \( m_1(cn) \) be the multiset of external messages in transit in \( cn \), or of messages occurring in an object in- or out-queue.

**Proposition 10.3.** If \( A_1(cn) \rightarrow cn' \) then
**Algorithm 1**: Stabilize routing and read internal link messages

**Input** Type 2 wellformed configuration $cn$ on a connected network graph

**Output** Configuration with stable routing and external link messages only

**repeat**

Use t-send on each proper link in $cn$ to broadcast routing tables to all neighbours ;

**repeat**

Use t-rcv to dequeue one message on a link in $cn$

**until** t-rcv no longer enabled ;

Use msg-rcv, msg-route, msg-delay-1, obj-rcv to dequeue one message from each link, if possible

**until** link queues contain only external messages, and routing is stable

Figure 10: Algorithm 1– Stabilize routing and empty link queues of internal messages

1. $\text{graph}(cn) = \text{graph}(cn')$
2. $cn'$ has stable routing
3. $cn'$ has external link messages only
4. $t(cn') = t_1(cn)$
5. $o(cn') = o_1(cn)$
6. $m(cn') = m_1(cn)$

**Proof.** Property 1 and 2 are immediate. Property 3 and 4 can be read out of the termination proof. For the remaining three properties observe first that $t_1$, $o_1$, and $m_1$ are all invariant under the transitions used in algorithm 1. The equations follows by noting that only external messages (and so no object closures) are in transit in $cn'$. □

Prop. 10.3 shows the “almost uniqueness” property alluded to above. The normal form property suggested by prop. 10.3 motivates a notion of equivalence “up to stabilization” defined below.

**Definition 10.4** ($\equiv_1$).

1. Let $cn_1 \ R_1 \ cn_2$ if and only if $cn_1$ and $cn_2$ are both type 2 well-formed, $\text{graph}(cn_1) = \text{graph}(cn_2)$, $z_1 = z_2$, $t_1(cn_1) = t_1(cn_2)$, $o_1(cn_1) = o_1(cn_2)$, and $m_1(cn_1) = m_1(cn_2)$.

2. Let $cn_1 \equiv_1 cn_2$ if there are $cn_1', cn_2'$ such that

$$A_1(cn_1) \leadsto_{A_1} cn_1' \ R_1 cn_2' \leadsto_{A_1} A_1(cn_2)$$

Prop. 10.3 together with termination of $A_1$ allows the existential quantifiers in def. 10.4.2 to be exchanged by universal ones.
Algorithm 2: Normalization

Input Type 2 well-formed configuration $cn$ on a connected network graph

Output Configuration in type 2 normal form

fix a NID $u$ ;
run alg. 1 ;
repeat
while some object queue is nonempty {
    use msg-send, msg-delay-2, call-rcv to dequeue one message from each
    nonempty object queue } ;
while an object exists not located at $u$ {
    use obj-send to send the object towards $u$ } ;
run alg. 1
until all objects are located at $u$ and queues contain only external messages

Figure 11: Algorithm 2 – Normalization

Corollary 10.5. If $A_1(cn) \rightsquigarrow cn'$ then $cn \equiv_1 cn'$
Proof. We have $A_1(cn) \rightsquigarrow cn' \mathcal{R} cn' \rightsquigarrow A_1(cn')$. □

Lemma 10.6. $\equiv_1$ is reduction closed
Proof. See appendix A. □

Proposition 10.7. $\equiv_1$ is a type 2 witness relation
Proof. See appendix A. □

Corollary 10.8. If $A_1(cn) \rightsquigarrow cn'$ then $cn \simeq cn'$
Proof. By prop. 10.7 and corollary 10.5. □

10.2 Normalization

When then turn to the second normalization step, to empty object queues and migrate all object closures to a central node. The normalization procedure is algorithm 2 shown in fig. 11. Let $A_2(cn) \rightsquigarrow cn'$ if $cn'$ is a possible result of applying algorithm 2 to $cn$. Initially a node $u_0$ is chosen towards which all objects will migrate during normalization. Normalization is performed in cycles, each cycle starting and ending in a stable configuration. In each cycle first object in- and out-queues are emptied. Then, objects not yet at $u_0$ are migrated one step toward $u_0$. Routing is not needed for this. It is sufficient to know that migration toward $u_0$ is possible.

Proposition 10.9. Algorithm 2 terminates
Proof. See appendix A. □

We then turn to normal forms and define first a couple of auxiliary operations. Let $t_2(cn)$ be the multiset of method containers $tsk = t(o,l,s)$ such that one of the following cases apply:
• tsk is a task container in cn.
• There is a message call \((a', o, m, v)\) in transit, \(o \not\in \text{Ext}\), \(l = \text{locals}(o, m, v)\) and \(s = \text{body}(o, m)\).

Let \(o_2(cn)\) be the multiset of object containers \(o(o, a, u, \epsilon, \epsilon)\) for which the following apply:
• \(u = u_0\)
• There is an object container \(\text{obj} = o(o', a', u', q_{in}, q_{out}) \preceq cn\)
• \(a'(x) = a(x)\) for all variables \(x\)

Also say that \(cn\) has external messages only, if object queues are empty and link queues in \(cn\) contain only external messages.

**Definition 10.10 (Normal Form).** A well-formed configuration \(cn\) is in normal form, if
1. \(cn\) has stable routing
2. \(cn\) has external messages only
3. \(t(cn) = t_2(cn)\)
4. \(o(cn) = o_2(cn)\)
5. \(m(cn) = m_1(cn)\)

**Proposition 10.11.** Suppose \(cn\) is well-formed. If \(A_2(cn) \leadsto cn'\) then
1. \(cn'\) is in normal form
2. \(\text{graph}(cn) = \text{graph}(cn')\)
3. \(t_2(cn) = t(cn')\)
4. \(o_2(cn) = o(cn')\)
5. \(m_1(cn) = m(cn')\)

**Proof.** See appendix A. \(\square\)

Similar to prop. 10.4, prop. 10.11 justifies a notion of normal form equivalence as follows.

**Definition 10.12 (≡_2).** 1. Let \(cn_1 \preceq_2 cn_2\) if and only if \(cn_1\) and \(cn_2\) are both well-formed, \(\text{graph}(cn_1) = \text{graph}(cn_2), t_2(cn_1) = t_2(cn_2), o_2(cn_1) = o_2(cn_2),\) and \(m_1(cn_1) = m_1(cn_2)\).
2. Let \(cn_1 \equiv_2 cn_2\) if and only if there are \(cn'_1, cn'_2\) such that

\[ A_2(cn_1) \leadsto cn'_1 \quad R_2 \quad cn'_2 \leadsto A_2(cn_2) \]

Clearly, \(\equiv_2\) identifies more extended configurations than \(\equiv_1\).
Corollary 10.13. \( \equiv_1 \subseteq \equiv_2 \)

Proof. If \( t_1(cn_1) = t_1(cn_2) \) then \( t_2(cn_1) = t_2(cn_2) \) and similar for \( o_1 \) and \( o_2 \). The result follows.

We also obtain that normalization respects normal form equivalence.

Corollary 10.14. If \( A_2(cn) \rightsquigarrow cn' \) then \( cn \equiv_2 cn' \)

Proof. By prop. 10.11.

The proof of reduction closure follows that of lemma 10.6 quite closely.

Lemma 10.15. \( \equiv_2 \) is reduction closed.

Proof. See appendix A.

Proposition 10.16. \( \equiv_2 \) is a type 2 witness relation

Proof. Similar to the proof of prop. 10.7.

It follows that \( cn_1 \equiv_2 cn_2 \) implies \( cn_1 \simeq cn_2 \).

Corollary 10.17. If \( A_2(cn) \rightsquigarrow cn' \) then \( cn \simeq cn' \)

Proof. None of the rules used in alg. 2 affects the shape of the normal form. Thus, if \( A_2(cn) \rightsquigarrow cn' \) then \( cn \equiv_2 cn' \). But then \( cn \simeq cn' \), by prop. 10.16.

11 Correctness

The goal is to prove soundness and full abstraction of the network semantics, i.e. that for any two type 1 configurations \( \text{bind} \ o \ cn \), \( \text{bind} \ o' \ cn' \) in standard form, \( \text{bind} \ o \ cn \simeq \text{bind} \ o' \ cn' \) if and only if \( \text{down}(cn_1) \simeq \text{down}(cn_2) \).

However, since we have set up the semantics such that \( \simeq \) applies without modification at both type 1 and type 2 levels it suffices to prove that \( \text{bind} \ o \ cn \simeq \text{down}(cn) \).

To accomplish this we represent each type 1 configuration as a type 2 configuration in normal form. We first fix an underlying graph represented as a well-formed type 2 configuration \( cn_{\text{graph}} \) and a distinguished UID \( u_0 \) in this graph, similar to the way initial configurations are defined in section 6.

Thus, \( cn_{\text{graph}} \) consists of nodes and links only, each node \( u \) in \( cn_{\text{graph}} \) has the form \( (u, t) \), and each link has the form \( (u, \varepsilon, u') \). The routing tables \( t \) are defined later. Defining a suitable representation map is a little cumbersome. A first complication is that names in the type 1 semantics (which includes the binder) need to be related to names in the type 2 semantics, which does not include the binder; but on the other hand has different generator functions (the function \( \text{newo} \)). For external names this is not a problem, but for bound names some form of name representation map is useful to connect the two types of names. Accordingly, we fix an injective name representation map \( \text{rep} \), taking names \( o \) in the type 1 semantics to names \( \text{rep}(o) \) in the type 2 semantics. For convenience we extend the name representation map \( \text{rep} \) to
external names \( o \in \text{Ext} \) by \( \text{rep}(o) = o \), to arbitrary values by \( \text{rep}(p) = p \), to task environments by \( \text{rep}(l)(x) = \text{rep}(l(x)) \) and similarly for object environments. The only slight complication in defining the mapping \( \text{down} \) is that we need an operation to send a type 1 call container as a message in the type 2 semantics. This is done by the operation \( \text{send} \) which sends a call container originating at \( o \) onto object \( o:s \) output queue as follows:

\[
\text{send}(c(o, o', m, v), o(a, a, u, q_{in}, q_{out}) \ cn) = o(a, a, u, q_{in}, \text{enq}(\text{call}(o, o', m, v), q_{out})) \ cn
\]

We can then define the type 2 representation of the type 1 configuration bind \( o.cn \) (leaving routing tables to be defined shortly) as the extended configuration \( \text{down}(cn)(cn_{graph}) \) where \( \text{down} \) is defined by induction on the structure of \( cn \) as follows:

- \( \text{down}(0)(cn) = cn \)
- \( \text{down}(cn_1 cn_2) = \text{down}(cn_1) \circ \text{down}(cn_2) \)
- \( \text{down}(t(o, l, s))(cn) = t(\text{rep}(o), \text{rep}(l), s) \ cn \)
- \( \text{down}(a(o, a))(cn) = a(\text{rep}(o), \text{rep}(a), u_0, \varepsilon, \varepsilon) \ cn \)
- \( \text{down}(c(o, o', m, v))(cn) = \text{send}(c(\text{rep}(o), \text{rep}(o'), m, \text{rep}(v)), cn) \)

In other words, we represent type 1 configurations by first assuming some underlying network graph, and then mapping the containers individually to type 2 level. The only detail remaining to be fixed above is the routing tables. For \( n_0 \) the initial routing table, \( t_0 \), needs to register all objects in \( cn_0 \), i.e.

\[
t_0 = \text{reg}(g(o_0), u_0, \text{reg}(g(o_1), u_0, \text{reg}(\cdots, \text{reg}(g(o_n), u_0, \bot))) \cdots)
\]

where \( o_0, \ldots, o_n \) are the OID's in \( cn'_0 \). For nodes \( n(u, t) \) where \( u \neq u_0 \) we let \( t \) be determined by some stable routing. This is easily computed using alg. 1, and we leave out the details. This completes the definition of \( \text{down}(cn) \).

**Lemma 11.1** (Up and Down Property). Let bind \( z.cn \) be type 1 well-formed in standard form.

1. If bind \( z.cn \rightarrow \text{bind} z'.cn' \) then \( \text{down}(cn) \rightarrow^* \circ \simeq \text{down}(cn') \)
2. If \( \text{down}(cn) \rightarrow cn'' \) then for some \( z', cn' \), bind \( z.cn \rightarrow^* \text{bind} z'.cn' \) and \( cn'' \simeq \text{down}(cn') \)

**Proof.** See appendix B. \( \square \)

We can now prove the main result of this first part of the paper.

**Theorem 11.2** (\( \mu\text{ABS} \) Implementation Correctness). For all wellformed type 1 configurations \( cn \) on connected network graphs, \( cn \simeq \text{down}(cn) \)
Proof. Define \( \mathcal{R} \) by
\[
\mathcal{R} = \{ (cn, cn') \mid \text{down}(cn) \simeq cn' \}
\] (4)

We show that \( \mathcal{R} \) is a witness relation.

First for reduction-closure in both directions: If \( cn_1 \mathcal{R} cn_2 \) then \( \text{down}(cn_1) \simeq cn_2 \). If \( cn_1 \rightarrow cn_1' \) then by the Up and Down Property, 1, \( \text{down}(cn_1) \rightarrow^* cn' \simeq \text{down}(cn_1') \). We get that \( cn_2 \rightarrow^* cn_2' \) such that \( cn' \simeq cn_2' \). But then \( cn_1' \mathcal{R} cn_2' \).

Conversely, if \( cn_2 \rightarrow cn_2' \) then \( \text{down}(cn_1) \rightarrow^* cn' \) and \( cn' \simeq cn_2' \). By the Up and Down Property, \( cn_1 \rightarrow^* cn_1' \) and \( cn' \simeq \text{down}(cn_1') \). But then \( cn_1' \mathcal{R} cn_2' \) as desired.

Barb Preservation is very direct, also in both directions: Assume \( cn_1 \mathcal{R} cn_2 \).
Then \( cn_1 \downarrow \text{obm}(v) \) if and only if \( \text{down}(cn_1) \downarrow \text{obm}(v) \) if and only if \( cn_2 \downarrow \text{obm}(v) \). This completes the proof.

12 Discussion

We have presented a sound and fully abstract semantics for a rudimentary object language, in terms a network-based execution model. Thanks in part to a novel explicit mixing of messaging and routing we are able to present the model at a level where it could in principle be implemented in a provably correct fashion directly on top of silicon, or integrated in a hypervisor such as Xen [3], assuming reliable link layer functionality only.

Soundness and full abstraction is a useful validation that the network semantics induces the same behaviour on \( \mu \text{ABS} \) programs as the reference semantics. The network semantics, however, lacks a scheduler to determine e.g. when to migrate objects and how to schedule threads on single nodes. Such a scheduler will resolve nondeterministic choices left open in the network semantics presented here. Once such a scheduler is added, soundness and full abstraction is lost. We can, however, easily adapt the results presented in this paper to a notion of barbed simulation, obtained by instead of requiring preservation of observations and reductions in both directions, requiring preservation only in one. Then correctness for barbed simulation, that \( \text{down}(cn) \) simulates bind \( o.cn \), is obtained as a corollary.

Substantial work has been going on in the HATS project on the ABS language [22] and its extensions, for instance towards software product lines [35]. Johnsen et al [23] suggests an extension of ABS with deployment components for resource management. We are mainly interested in the \( \mu \text{ABS} \) language as an example. Essentially, however, our work is language independent, and we could apply the approach presented here to a version of core Erlang with minor changes only. Some details would be different, in particular the treatment of Erlang’s pattern match-based message reception construct. The changes, however, would be local only, and so make little essential difference.

Much work has been done on object/component mobility in the \( \pi \)-calculus tradition [29], and on the implementation of high-level object or process-oriented languages in terms of more efficiently implementable low level cal-
culi. In [36], following earlier work on Pict [31], Fournet’s distributed join-calculus [16], and the JoCaml programming language [9], a compiler is implemented and proved correct for Nomadic Pict, a prototype language with very similar functionality to our μABS language: principally asynchronous message passing between named, location-oblivious processes. The target language extends Pierce and Turner’s Pict language with synchronous local communication and asynchronous message passing between located processes. In comparison with [36] the use of name-based routing allows us to use barbed equivalence in place of coupled simulation [30] and as a consequence obtain a simpler correctness proof, due to the need for locking in the central forwarding server scheme used in [36]. JoCaml also uses forward chaining, along with an elaborate mechanism to collapse the forwarding chains. In the Klaim project [4] compilers were implemented and proved correct for several variants of the Klaim language, using the Linda tuple space communication model and a centralized name server to identify local tuple servers. The Oz kernel language [38] uses a monotone shared constraint store in the style of concurrent constraint programming. The Oz/K language [27] adds to this a notion of locality with separate failure and mobility semantics, but no real distribution or communication semantics is given (long distance communication is reduced to explicit manipulation of located agents, in the style of Ambient calculus [7]).

Our correctness proof uses reduction semantics and barbed equivalence. This is rather standard in the process algebra literature, cf. [15, 8, 18]. Both Sewell et al [36] and Fournet et al [17] use coupled simulation in order to handle problems related to preemptive choice. This complication does not arise in our work, whence barbed equivalence suffices. However, barbed equivalence is mostly useful for closed systems modeling where the stimuli to which an observed system is to be exposed must be given up front, as part of the initial configuration. A structural account based on some form of contextual equivalence [32], or on bisimulation equivalence along with a labelled transition semantics instead, would be more suitable. Work in this direction is currently going on. Replacing our reference reduction-based semantics with a labelled transition semantics is fairly straightforward. The bigger challenge is to develop a suitable structural account at the network level, allowing partially defined configurations to be composed.

Another direction for future work is to extend the μABS language. Experiments in this directions are going on. In [11] we extend μABS with futures (aka promises [28]) asplaceholders for return values. This extension turns out to be not at all trivial. In other directions it is of interest to consider models with node power-on/power-off, in order to model system with adaptive power consumption, as well as various forms of node failure, along similar lines as [14]. The model can be used as a platform for language-based studies of load balancing and resource adaptation. We have extended the network semantics reported here to the full core ABS language [22] and implemented a multi-core simulation engine. This work is reported in [10]. There we show how the network semantics presented here can be split into a language interpreter layer and a language inde-


and technologies, CoNEXT ’09, pages 1–12, New York, NY, USA, 2009. ACM.


[31] B. C. Pierce and D. N. Turner. Pict: A programming language based on
the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, Proof,
Language and Interaction: Essays in Honour of Robin Milner, pages

and J. Rutten, editors, Advanced Topics in Bisimulation and Coinduc-
tion, volume 52 of Cambridge Tracts in Theoretical Computer Science,

[33] D. Sangiorgi. Introduction to Bisimulation and Coinduction. Cam-

[34] D. Sangiorgi, N. Kobayashi, and E. Sumii. Environmental bisimula-
tions for higher-order languages. ACM Trans. Program. Lang. Syst.,


[36] P. Sewell, P. T. Wojciechowski, and A. Unyapoth. Nomadic pict: Pro-
gramming languages, communication infrastructure overlays, and sem-
antics for mobile computation. ACM Transactions on Programming
Languages and Systems, 34, April 2010.

[37] A. Singla, P. B. Godfrey, K. Fall, G. Iannaccone, and S. Ratnasamy. Scal-
able routing on flat names. In Proceedings of the 6th International
Conference, Co-NEXT ’10, pages 20:1–20:12, New York, NY, USA,
2010. ACM.

[38] G. Smolka. The definition of kernel oz. In Andreas Podelski, editor,
Constraints: Basics and Trends, volume 910 of Lecture Notes in Com-


of the thirteenth annual ACM symposium on Parallel algorithms and
architectures, SPAA ’01, pages 1–10, New York, NY, USA, 2001. ACM.

[41] M. van Steen, F. J. Hauck, G. Ballintijn, and A. S. Tanenbaum. Algo-
Rithmic design of the globe wide-area location service. Comput. J.,
Appendix A: Proofs for Section 10

Proposition 10.2  Algorithm 1 terminates

Proof. In each iteration of the outermost loop of alg. 1, exactly one message is enqueued on each proper link, and at least one message is dequeued (from all link queues), so the sum of messages in transit in link queues does not exceed its initial value. The rules msg-rcv, msg-delay-1, obj-rcv cause messages to leave the link queues, except for external messages, which are copied onto the self-loop queues. If the link queues have only routing table messages the algorithm terminates in that iteration. So if the algorithm fails to terminate it must be because msg-route is from some point $n_0$ onwards applied in each iteration of the outermost loop. From $n_0$ onwards, no messages other than table updates are delivered (to the receiving node, or to the receiving object). In particular, no object messages can be in transit on a link from that point onwards. We show that then routing tables must at some point stabilize. At point $n_0$ (as all other points) each node $u$ has $t(o) = (u, 0)$ whenever $o$’s host is $u$, by def. 8.1.7. Let $m_0$ be the length of the largest link queue at the point from which no messages are delivered. After $n_0 + m_0 + 1$ iterations, each node $u$ has received at least one table update from each of its neighbours $u'$, and the last table update applied to $u$ has $t(o) = 0$. As result, at point $n_0 + m_0 + 1$ each node $u$ has $t(o) = (u', 1)$ whenever the host of $o$ is $u'$ and the minimal length path from $u$ to $u'$ has length 1. The entry of $u$’s routing table for $o$ will not change from that point onwards. We say that those entries are stable. Proceeding, let $m_1$ be the length of the largest link queue at at point $n_0 + m_0 + 1$. After $n_0 + m_0 + 1 + m_1 + 1$ iterations each routing table entry with length 2 (or less) will be stable. In the limit each entry will be stable. It follows that algorithm 1 must terminate, since, once routing has stabilized, rule msg-route-ext can only be applied a finite number of times before the message will be delivered.

The only detail remaining to be checked is that a message can always be read from a link, but table and object messages can always be delivered, and call and future messages can also always be delivered, if nothing else to the self loop, in case the routing table has not yet been updated, or if the message is external and the destination object is not known to the routing table. This is the only case where msg-delay-1-ext is used, in fact. This completes the argument.  

Lemma 10.6  $\equiv_1$ is reduction closed

Proof. Assume that $cn_1 \rightarrow cn_1'$ and $cn_1 \equiv_1 cn_2$. We find $cn_2'$ such that $cn_2 \rightarrow^* cn_2'$ and $cn_1' \equiv_1 z_2' : cn_2'$. The proof is by cases on the rewriting rule applied. The details are a bit messy, but straightforward. For rules not among call-send, call-rcv, if

$$A_1(cn_1) \rightsquigarrow cn_{1,1} R_1 cn_{2,1} \rightsquigarrow A_1(cn_2)$$

(5)

and $A_1(cn_1') \rightsquigarrow cn_{1,1}'$ then we obtain $cn_{1,1}' R_1 cn_{1,1}$ by prop. 10.3 and since the correspondences between tasks, objects, and call and future messages

35
are maintained in pre- and poststates. For the two remaining rules, we proceed:
call-send: We may assume that
\[ cn_1 = cn_{1,1} o(a, u, q_{in}, q_{out}) t(o, l, x = e_1 ! m(e_2); s) \]  
\[ cn'_1 = cn_{1,1} o(a, u, q_{in}, \text{enq}(msg, q_{out})) t(o, l, s) \]
where \( v = e_2(a, l) \), \( msg = \text{call}(o, o', m, v) \), \( o' = e_1(a, l) \). We get
\[ \mathcal{A}_1(cn_1) \rightsquigarrow cn''_1 R cn''_2 \rightsquigarrow \mathcal{A}_1(cn_2) \]
for some choice of \( cn''_1 \), etc. By prop. 10.3,
\[ o(a, u, q_{in}, q_{out}) t(o, l, x = e_1 ! m(e_2); s) \in cn''_1 \]
and hence, by the definition of \( R \),
\[ o(a, u, q_{in}, q_{out}) t(o, l, x = e_1 ! m(e_2); s) \in cn''_2 \]
as well. But then it follows that the configuration \( cn_2 \) can mimic the call-send step by \( cn_1 \) by first stabilizing to \( cn''_2 \) and then performing the call-send step, obtaining \( cn'_2 \).
call-rcv: Can be proved by the same strategy.

**Proposition 10.7** \( \equiv_1 \) is a type 2 witness relation

**Proof.** Symmetry is immediate, and reduction closure follows by lemma 10.6. For barb preservation, from \( cn_1 \equiv_1 cn_2 \) we get
\[ \mathcal{A}_1(cn_1) \rightsquigarrow cn''_1 R cn''_2 \rightsquigarrow \mathcal{A}_1(cn_2) . \]
If \( cn_1 \Downarrow o ! m(v) \) then \( cn_1 \) has the shape \( cn''_1 o(a, u, q_{in}, q_{out}) \) and \( \text{hd}(q_{out}) = \text{call}(o', a, m, v) \). By prop. 10.3 and the details of alg. 1, since \( o \) is external and not routable, the message call \( (o', a, m, v) \) occurs in the self-loop queue on \( u \).
By the definition of \( \equiv_1 \), call \( (o', a, m, v) \) occurs in a self-loop of \( cn'_2 \). A few reductions exposes call \( (o', a, m, v) \) at the head of that queue, at configuration \( cn''_2 \), say. Then \( cn'_2 \rightarrow^{*} \) \( cn''_2 \) and \( \mathcal{A}_1(cn_2) \rightarrow^{*} \mathcal{A}_1(cn_2) \) we obtain \( cn_2 \Downarrow o ! m(v) \) and the proof is complete.

**Proposition 10.9** Algorithm 2 terminates

**Proof.** Routing is stable after each run of alg. 1, and none of the rules applied in the first inner loop affect routing stability. Also, after the first run of alg. 1, links contain only external calls. Whenever an object out-queue is nonempty, one of msg-send or msg-delay-2 will be enabled. By Buffer Cleanliness, call-rcv will be applicable if the object in-queue is nonempty, decreasing in-queue size by one. Thus, when the first while loop is exited, object queues are empty. The second while terminates when all objects not yet at \( u_0 \) have been put on the wire. At the end of each outer loop, routing is stabilized and link queues emptied (except for external messages).
Once emptied, out-queues remain empty. In-queues may contain messages at the start of the second iteration, but after that, only external messages remain in either link or object queues, except for object closures, which are consumed once they reach $u_0$.

**Proposition 10.11** Suppose $cn$ is well-formed. If $A_2(cn) \rightsquigarrow cn'$ then

1. $cn'$ is in normal form
2. $\text{graph}(cn) = \text{graph}(cn')$
3. $t_2(cn) = t(cn')$
4. $o_2(cn) = o(cn')$
5. $m_1(cn) = m(cn')$

**Proof.** Property 10.11.2 follows from prop. 10.3.1. For property 10.11.3 observe first that the function $t_2$ is invariant under transitions used in alg. 2. On termination of alg. 2 only external messages are in transit, and since no rule causes a task to be modified, 10.11.3 follows.

For 10.11.4 let $o(o_2, a_2, u_2, q_{in.2}, q_{out.2}) \in o(cn')$. We need to show that

$$o(o_2, a_2, u_2, q_{in.2}, q_{out.2}) \in o_2(cn).$$

By definition, $q_{in.2} = q_{out.2} = \epsilon$. Also, $u_2 = u_0$. We know that there is an object container $o(o, a', u', q_{in}, q_{out}) \models cn$, as there is a 1-1 correspondence between object containers in pre- and poststate for each transition used in alg. 2. We also know that $a'(x) = a_2(x)$ for all $x$.

For 10.11.5 the property holds as it does so already for alg. 1.

We finally need to prove 10.11.1. Property 10.10.1 is trivial, as each run of alg. 2 ends with a run of alg. 1, and alg. 1 ensures that $cn'$ has stable routing. Property 10.10.2 holds since alg. 1 ensures external link messages only, and since on termination, alg. 2 ensures empty object queues. For 10.10.3 the result follows since only external messages are in transit in $cn'$. For 10.10.4, if $obj = o(o, a, u, \epsilon, \epsilon)$ satisfies the properties defining $o_2$ above then, referring to those conditions, $u = u' = u_0$, $a' = a$, $q_{in} = \epsilon = q_{out}$, and $obj \in cn'$ as needed to be shown. Finally, 10.10.5 holds since it does so already for alg. 1.

**Lemma 10.15** $\equiv_2$ is reduction closed.

**Proof.** Assume that $cn_1 \rightarrow cn'_1$ and $cn_1 \equiv_2 cn_2$. We find $cn'_2$ such that $cn_2 \rightarrow^* cn'_2$ and $cn'_1 \equiv_2 cn'_2$. As above the proof is by cases on the rewrite rule. We can assume that $cn_2$ is in normal form, by 12 and transitivity of $\equiv_2$. For rules that do not affect $t_2(cn_1)$, $o_2(cn_1)$, or $m_1(cn_1)$ the result is trivial. Rules in fig. 5 commute directly, i.e. the same rule applied to $cn_1$ can be applied to $cn_2$, in the same way. This follows since $cn_2$ is in normal form. Rules such as msg-send, msg-rcv that ship around messages between object and link
queues are also very easy to prove, by reference to prop. 10.11. For the remaining cases:
call-send: We may assume that
\[ cn_{1} = cn_{1,1} o(a, a, u, q_{in}, q_{out}) t(o, l, x = e_{1!m}(e_{2}); s) \] (12)
\[ cn'_{1} = cn_{1,1} o(a, a, u, q_{in}, enq(msg, q_{out})) t(o, l, s) \] (13)

where \( v = e_{2}(a, l) \), \( msg = call(o, a', m, v) \), \( o' = c_{1}(a, l) \), and \( o' \notin Ext \). By 10.11, since \( cn_{2} \) is in normal form, we obtain
\[ A_{2}(cn_{1}) \Rightarrow cn''_{1} R_{2} cn_{2} \] (14)

for some choice of \( cn''_{1} \). By prop. 10.11 we obtain that
\[ cn_{2} = cn_{2,1} o(a, a', u, \varepsilon, \varepsilon) t(o, l, s) \] (15)

where \( a'(x) = a(x) \) for variables \( x \). It follows that \( cn'_{2} \) can be chosen so that
\[ cn'_{2} = cn_{2,1} o(a, a', u, \varepsilon, enq(msg, \varepsilon)) t(o, l, s), \] (16)

and
\[ A_{2}(cn'_{1}) \Rightarrow cn_{1,3} \] (17)

with \( cn_{1,3} = cn_{1,4} o(a, a', u, \varepsilon, enq(msg, \varepsilon)) t(o, l, s) \) such that \( cn_{1,4} R_{2} cn_{2,1} \). It follows that \( cn_{2} \rightarrow cn'_{2} \) and \( cn'_{1} \equiv_{2} cn'_{2} \), as desired.
call-rcv: In this case we get
\[ cn_{1} = cn_{1,1} o(a, a, u, q_{in}, q_{out}) \] (18)
\[ cn'_{1} = cn_{1,1} o(a, a, u, deq(q_{in}), q_{out}) t(o, locals(o, m, v), body(o, m)) \] (19)

where \( hd(q_{in}) = call(o', o, m, v) \). Again using prop. 10.11 with \( cn_{2} \) in normal form we get that
\[ A_{2}(cn_{1}) \Rightarrow cn''_{1} R_{2} cn_{2} \] (20)

for some choice of \( cn''_{1} \), and \( cn''_{1} \) can be written as
\[ cn''_{1} = cn''_{1'} o(a', a', u, \varepsilon, \varepsilon) t(o, locals(o, m, av), body(o, m)) \] (21)

where \( a' \) is as in the previous case. Now using prop. 10.11 we obtain
\[ cn_{2} = cn_{2,1} o(a, a', u, \varepsilon, \varepsilon) t(o, locals(o, m, av), body(o, m)) \] (22)

and \( cn''_{1} R_{2} cn_{2} \), completing the case.
The remaining cases ret-2, get-2, new-2, and the object migration rules are proved in a similar fashion as the above.

\[ \square \]

**Appendix B: Proofs for Section 11**

**Lemma 11.1** (Up and Down Property). Let \( z.cn \) be type 1 well-formed in standard form.
1. If $\text{bind } z.cn \rightarrow \text{bind } z'.cn'$ then $\text{down}(cn) \rightarrow^* o \simeq \text{down}(cn')$

2. If $\text{down}(cn) \rightarrow cn''$ then for some $z'$, $cn'$, $\text{bind } z.cn \rightarrow^* \text{bind } z'.cn'$ and $cn'' \simeq \text{down}(cn')$

Proof. 1. First note that each transition in fig. 5 immediately translates into a corresponding transition at type 2 level, and moreover, the resulting type 2 configuration is in normal form. For the remaining transitions we proceed by cases:

**call:** As $cn$ is type 1 well-formed we can write $cn$ in standard form as

$$cn = \text{bind } z.cn_1 o(o,a) t(o,l,x = e_1!m(e_2); s),$$  \hspace{1cm} (23)

and $cn'$ as

$$cn' = \text{bind } z.\text{bind } f.cn_1 o(o,a) o(o',a') t(o,l,s)$$
$$t(o', \text{locals}(o', m, e_2(a,l)), \text{body}(o', m))$$

where $o' = e'_1(a,l)$. Let $v = \hat{e}_2(a,l)$. Fix $cn_{\text{graph}}$ and $u_0$ as above. We get

$$\text{down}(cn) = \text{down}(cn_1)(cn_{\text{graph}}) o(rep(o), rep(a), u_0, \varepsilon, \varepsilon)$$
$$o(rep(o'), rep(a'), u_0, \varepsilon, \varepsilon)$$
$$t(rep(o), rep(l), x = e_1!m(e_2); s)$$
$$\rightarrow \text{down}(cn_1)(cn_{\text{graph}}) o(rep(o), rep(a), u_0, \varepsilon, \varepsilon)$$
$$\text{enq}((\text{call}(rep(o'), m, rep(v)), \varepsilon))$$
$$o(rep(o'), rep(a'), u_0, \varepsilon, \varepsilon) t(rep(o), rep(l), s)$$
$$\rightarrow^* o \simeq \text{down}(cn_1)(cn_{\text{graph}})$$
$$o(rep(o), rep(a), u_0, \varepsilon, \varepsilon)$$
$$o(rep(o'), rep(a'), u_0, \varepsilon, \varepsilon) t(rep(o), rep(l), s)$$
$$t(rep(o'), \text{locals}(rep(o'), m, e_2(rep(a), rep(l)))}, \text{body}(o', m))$$

(By normalization and corollary 10.17)

$$= \text{down}(cn_1)(cn_{\text{graph}}) o(rep(o), rep(a), u_0, \varepsilon, \varepsilon)$$
$$o(rep(o'), rep(a'), u_0, \varepsilon, \varepsilon) t(rep(o), rep(l), s)$$
$$t(rep(o'), \text{locals}(o', m, e_2(a,l))), \text{body}(o', m))$$

$$= \text{down}(cn')$$

**call-ext:** We can write $cn$ and $cn'$ as

$$cn = \text{bind } z.cn_1 o(o,a) t(o,l, x = e_1!m(e_2); s)$$
$$cn' = \text{bind } z.cn_1 o(o,a) t(o,l,s) c(o,o', m, v)$$
where \( o' = e_1(a, l) \in \text{Ext} \) and \( v = e_2(a, l) \). We obtain:

\[
\text{down}(cn) = \text{down}(cn_1)(cn_{\text{graph}}) \circ (\text{rep}(o), \text{rep}(a), u_0, \varepsilon, \varepsilon) \\
\quad \circ (\text{rep}(o), \text{rep}(l), x = e_1!m(e_2); s) \\
\rightarrow \text{down}(cn_1)(cn_{\text{graph}}) \circ (\text{rep}(o), \text{rep}(a), u_0, \varepsilon, \varepsilon) \\
\quad \circ \text{enq}(\text{call}(\text{rep}(o'), m, \text{rep}(v)), \varepsilon) \circ t(\text{rep}(o), \text{rep}(l), s) \\
= \text{down}(cn')
\]

\textbf{new: Let}

\[
\begin{align*}
\text{cn} &= \text{bind } z.\text{cn}_1 \circ (o, a) \circ t(o, l, x = \text{new } C(e); s) \\
\text{cn}' &= \text{bind } z.\text{bind } o'.\text{cn}_1 \circ (o, a) \circ t(o, l[o'/x], s) \circ o(o', \text{init}(C, e(a, l)))
\end{align*}
\]

We can write \( \text{down}(cn_1)(cn_{\text{graph}}) \) as \( cn'_1 n(u_0, t_0) \). Also let \( o'' = \text{new } o(u_0) \).

\textbf{Note that \( o'' = \text{rep}(o') \). We calculate:}

\[
\text{down}(cn) = \text{cn}'_1 n(u_0, t_0) \circ (\text{rep}(o), \text{rep}(a), u_0, \varepsilon, \varepsilon) \circ t(\text{rep}(o), \text{rep}(l), x = \text{new } C(e); s) \\
\rightarrow \text{cn}'_1 n(u_0, \text{reg}(\text{rep}(o'), u_0, t_0)) \circ (\text{rep}(o), \text{rep}(a), u_0, \varepsilon, \varepsilon) \\
\quad \circ t(\text{rep}(o), \text{rep}(l)[\text{rep}(o'/x], s)] \\
\quad \circ o(\text{rep}(o'), \text{init}(C, e(a, l))) \circ o(o', \text{init}(C, e(a, l))) \circ u_0, \varepsilon, \varepsilon) \\
= \text{cn}'_1 n(u_0, \text{reg}(o', u_0, t_0)) \circ (\text{rep}(o), \text{rep}(a), u_0, \varepsilon, \varepsilon) \circ t(\text{rep}(o), \text{rep}(l[o'/x], s)] \\
\quad \circ o(\text{rep}(o'), \text{rep}(\text{init}(C, e(a, l)))) \circ u_0, \varepsilon, \varepsilon) \\
= \text{down}(cn')
\]

2. We proceed by cases on the type \( 2 \) rule applied to derive \( \text{down}(cn) \rightarrow cn'' \).

Rules in fig 5 are immediate since in those cases \( cn' \) can be found such that \( \text{down}(cn') = cn'' \). For rules among t-send, t-rcv, msg-send, msg-rcv, msg-route, msg-delay-1, msg-delay-2, call-rcv, obj-send, obj-rcv, using prop. 10.11, def. 10.12, and prop. 10.16 we can choose \( o' = o \) and \( cn' = cn \). For the remaining two rules:

\textbf{call-send:} We get that \( cn, \text{down}(cn) \) and \( cn'' \) have the shapes

\[
\begin{align*}
\text{cn} &= \text{cn}_1 \circ (o, a) \circ t(o, l, e_1!m(e_2); s) \\
\text{down}(cn) &= \text{down}(cn_1)(cn_{\text{graph}}) \circ (\text{rep}(o), \text{rep}(a), u, \varepsilon, \varepsilon) \\
&\quad \circ t(\text{rep}(o), \text{rep}(l), e_1!m(e_2); s) \\
\text{cn}'' &= \text{down}(cn_1)(cn_{\text{graph}}) \circ (\text{rep}(o), \text{rep}(a), u, \varepsilon, \varepsilon) \\
&\quad \circ \text{enq}(\text{call}(\text{rep}(o), \text{rep}(o'), m, \text{rep}(v)), \varepsilon) \circ t(\text{rep}(o), \text{rep}(l), s)
\end{align*}
\]

where \( e_1(a, l) = o' \) and \( e_2(a, l) = v \). We get two cases depending on whether \( o' \in \text{Ext} \) or not. Suppose first the latter. Then \( cn_1 \) can be written as \( cn'_1 \circ (o', a') \), and we can pick

\[
\text{cn}' = \text{cn}'_1 \circ (o, a) \circ (o', a') \circ t(o, l, s) \circ t(o', \text{locals}(o', m, v), \text{body}(o', m))
\]
We then use prop. 10.11 to conclude that $\text{down}(cn') \simeq cn''$, as desired. In case $o' \in \text{Ext}$ we get instead

$$cn' = cn_1 \circ (o,a) \triangleright (o,l,s) \circ (o,o',m,v)$$

and as above, $\text{down}(cn_1) = cn''$

new-2: We find $cn$, $\text{down}(cn)$ and $cn''$ as above of the shapes

$$cn = cn_1 \circ (o,a) \triangleright (o,l,x = \text{new}(e); s)$$
$$\text{down}(cn) = cn_1' \circ (u_0, t_0) \circ (\text{rep}(o), \text{rep}(a), u_0, \epsilon, \epsilon)$$
$$t(\text{rep}(o), \text{rep}(l), x = \text{new}(C); s)$$
$$cn'' = cn_1' \circ (u_0, \text{reg}(\text{rep}(o'), u_0, t_0)) \circ (\text{rep}(o), \text{rep}(a), u_0, \epsilon, \epsilon)$$
$$t(\text{rep}(o), \text{rep}(l)[\text{rep}(o')/x], s)$$
$$\circ (\text{rep}(o'), \text{init}(C, v), u_0, \epsilon, \epsilon)$$

where $o' = \text{new}(u_0), \hat{e}(a, l) = v$, and $\text{down}(cn_1)(cn_{\text{graph}}) = cn_1' \circ (u_0, t_0)$. We pick

$$cn' = cn_1 \circ (o,a) \triangleright (o,l'[x], s) \circ (o', \text{init}(C, v))$$

and get $\text{down}(cn') = cn''$. $\square$
Appendix C

Paper 2: Efficient and Fully Abstract Routing of Futures in Object Network Overlays
Efficient and Fully Abstract Routing of Futures in Object Network Overlays

Mads Dam and Karl Palmskog
School of Computer Science and Communication
KTH Royal Institute of Technology

Abstract

In distributed object systems it is desirable to be able to migrate objects transparently between locations, for instance in order to support load balancing and efficient resource allocation. Existing approaches build complex message routing infrastructures, typically on top of IP, using e.g. message forwarding chains, or centralized object location servers. These solutions are costly in terms of complexity and overhead. In an earlier paper we have shown how location independent routing can be used to implement process overlays in a sound, fully abstract, and efficient manner on top of an abstract network of processing nodes connected by asynchronous point to point channels. The overlays considered in that work allowed only one type of message, with modest requirements on global consistency. In this paper we show how the approach can be generalized to more complex object overlays involving futures, essentially placeholders for method return values that need to be kept consistent and propagated correctly to all objects that eventually need them.

1 Introduction

The ability to transparently and efficiently relocate objects between processing nodes is a basic prerequisite for many tasks in large scale distributed systems, including tasks such as load balancing, resource allocation, and management. By freeing applications from the burden of resource management they can be made simpler, more resilient, and easier to manage, resulting in a lower cost for development, operation and management.

The key problem is how to efficiently handle object and task mobility. Since in a mobile setting objects location changes dynamically, some form of application level routing is needed for inter-object messages to reach their destination. Various approaches have been considered in the literature (cf. [34] for a survey). One common implementation strategy is to use some form of centralized, replicated, or decentralized object location register, either for forwarding or for address lookup cf. [14, 34, 1, 18]. This type of solution requires some form of synchronization to keep registers consistent with physical location, or else it needs to resort to some form...
of message relaying, or forwarding. Forwarding by itself is another main implementation strategy used in e.g. the Emerald system [25], or in more recent systems like JoCaml [10]. Other solutions exist such as broadcast or multicast search, useful for recovery or for service discovery, but hardly efficient as a general purpose routing device in large systems.

In general one would like a mechanism for object mobility with the following properties:

• Low stretch: In stable state, the ratio between actual and optimal route lengths (costs) should be small.

• Compactness: The space required at each node for storing route information should be small (sublinear in the size of the network).

• Self-stabilization: Even when started in a transient state, computations should proceed correctly, and converge to a stable state. Observe that this precludes the use of locks.

• Decentralization: Routes and next hop destinations should be computed in a decentralized fashion, at the individual nodes, and not rely on a centralized facility.

Existing solutions are quite far from meeting these requirements: Location registers (centralizes or decentralized) and pointer forwarding regimes both preclude low stretch, and the use of locks precludes self-stabilization.

In a precursor to this paper [11] we suggest that the root of the difficulties lies in a fundamental mismatch between the information used for search and identification (typically, object identifiers, OID's), and the information used for routing, namely host identifiers, typically IP numbers. If we were to route messages not according to the destination location, but instead to the destination object, it should be possible to build object network overlays which much better fit the desiderata laid out above. In [11] we show that this indeed appears to be true (even if the problem of compactness is left for future investigation). The key idea is to use a form of location independent (also known as flat, or name independent) routing [20, 2, 21] that allows messages (rpc’s) to be routed directly to the called object, independently of the physical node on which that object is currently executing. In this way a lot of the overhead and performance constraints associated with object mobility can be eliminated, including latency and bandwidth overhead due to looking up, querying, updating, and locking object location databases, and overhead due to increased traffic, for instance for message forwarding.

The language considered in [11] allows to define a dynamically growing or shrinking collection of objects communicating by asynchronous rpc, and thus its functionality is not much different from a core version of Erlang [5], or the nomadic PICT language studied in [34]. The question we raise in [11] is how program behaviour is affected by being run in the networked model, as compared with a more standard (reference) semantics given here using rewrite logic. This comparison is of interest, since the reference semantics is given at a high level of abstraction and ignores almost all aspects of physical distribution, such a location, routing, message passing, and so on.
we show that, with a maximally nondeterministic networked semantics, and in the sense of barbed equivalence [31] which is a standard equivalence to study in these types of applications [9, 16, 17], programs exhibit the same behaviour in both cases.

Messaging in [11] is very simple. The implicit channel abstraction used in the reference semantics is essentially that of a reliable, unordered communication channel. Messages (calls) are sent according to the program order, but the order in which they are acted upon is arbitrary\footnote{This is not strictly speaking true in general, as in the reference semantics, the program order on calls may induce happens-before constraints on external method calls that cannot be realized in the networked semantics because messages are explicitly queued and can always be shuffled. However, barbed equivalence is not sensitive to this type of happens-before constraint in the reference semantics.}. Soundness and full abstraction for the networked semantics is therefore an interesting and useful observation, since it allows many conclusions made at the level of abstract program behaviour to transfer to the setting of the networked realization.

The question is how sensitive these results are to the type of communication taking place at the abstract level. The overlays considered in the earlier work allows only one type of message, with modest requirements on global consistency. It is of interest to examine also languages with richer communication structures than asynchronous point to point message passing. To this end we enrich in this paper the language studied earlier, and show that the conclusions of our previous work remain valid, however with more involved constructions. The extension results in much more complex object overlays involving so-called future variables that need to be kept consistent and propagated correctly to all objects that may eventually need them.

Future variables [7, 15, 27, 36, 28, 13] are placeholder variables for values that may be waiting to get instantiated. Futures are used extensively in many concurrent and distributed high-level languages, libraries, and models including Java, .NET, scheme, concurrent LISP, Oz, to name just a few. Many versions of future variables exist in the literature. Our work uses futures as placeholders for return values of remote method calls, as in [6, 13, 8]. Other models exists, such as the transparent first-class futures considered in [3], or the concurrent constraint store model of e.g. Oz [35, 28].

Futures need a messaging infrastructure to propagate instantiations. Consider a remote method call \( x = obj.m(arg) \). The effect of the call is the creation of two items:

1. A remote thread evaluating \( obj.m(args) \)
2. The assignment of a future to \( x \). The future is initially uninstantiated, but is intended to become instantiated after the remote call has returned.

This allows long running tasks to be offloaded to a remote thread with the main thread proceeding to other tasks. When the return value is eventually
needed, the calling thread can request it by executing a `get` on the future. If `x` is uninstantiated, this causes the future to block.

The problem is that futures can be transmitted as parameters between threads as well. If `y` is a future occurring in `args`, there must be some means for the value eventually assigned to `y` to find its way to the remote thread computing `obj.m(args)`, either by forwarding the value after it becomes available, or by the remote thread querying either the caller or some centralized lookup server for the value of `y`, if and when it is needed. This creates very similar problems to those arising from object migration. Thus it would seem likely that location independent routing could benefit propagation of futures as well, and as we show in this paper, indeed this is so. In the case of futures, however, the problems are aggravated: In order for the networked implementation to be correct (sound and fully abstract) we must be able to show that future assignments are unique and propagate correctly to all objects needing the assignment, without resorting to solutions that are overly inefficient such as flooding.

Many strategies for future propagation exist in the literature [19, 30]. In this work we use eager forward chaining where assignments are propagated along the flow of futures as soon as they are instantiated. Other propagation strategies exist, including strategies that use various forms of location registers, and lazy strategies which look up futures only as needed. Either approach may benefit from the use of location independent routing.

Our main result is to show that, with a fully nondeterministic semantics, the abstract semantics and the networked semantics with futures implemented by eager forward chaining correspond in the sense of barbed equivalence. This is interesting in itself, as it shows that the networked semantics captures the abstract behaviour very accurately. Also it follows that, for the case when a scheduler is added (pruning some execution branches), a similar correspondence holds, but now for barbed simulation instead of barbed bisimulation.

The proof uses a normal form construction in two stages. First, it is shown that each well-formed configuration in the networked semantics can be rewritten into an equivalent form with optimal routes. The second stage of the normalization procedure then continues rewriting to a form where in addition all messages that can be delivered also are delivered, and where all objects are migrated to some central node. Correctness of the normalization procedure essentially gives a Church-Rosser like property, that transitions in the networked semantics commute with normalization. Normalization brings configurations in the networked semantics close to the form of the reference semantics, and this then allows the proof to be completed.

The paper is organized as follows: In section 3 we first introduce the mABS language syntax, and the network oblivious reference semantics of mABS is given in section 4. In section 5 we present type 1 barbed equivalence, the notion of barbed equivalence adapted to the reference semantics. Then, in section 6, we turn to the network semantics and present the runtime syntax and the reduction rules. We proceed by detailing the well-formedness conditions for the network semantics in section 7 and adapt
barbed equivalence to the network semantics in section 8. We then present the normal form construction in section 9 and continue by completing the correctness proof in section 10. In section 11 we discuss scheduling, and finally in section 12 we conclude. Longer proofs have been deferred to the appendix.

2 Notation

We use a standard boldface vector notation to abbreviate sequences, for compactness. Thus, \( x \) abbreviates a sequence \( x_0,\ldots,x_n \), possibly empty, and \( f(x) \) abbreviates a sequence \( f_1 x_1,\ldots,f_n x_n \), etc. Let \( x = x_1,\ldots,x_n \).

Then \( x_0, x \) abbreviates \( x_0,\ldots,x_n \). Let \( g : A \to B \) be a finite map. The update operation for \( g \) is \( g[\text{\( b/a \)}](x) = g(x) \) if \( x \neq a \) and \( g[\text{\( b/a \)}](a) = b \). We use \( \perp \) for bottom elements, and \( A \perp \) for the lifted set with partial order \( \sqsubseteq \) such that \( a \sqsubseteq b \) if and only if either \( a = b \in A \) or else \( a = \perp \). Also, if \( x \) is variable ranging over \( A \) we often use \( x_\perp \) as a variable ranging over \( A \perp \). For \( g \) a function \( g : A \to B \) we write \( g(a) \downarrow \) if \( g(a) \in B \), and \( g(a) \uparrow \) if \( g(a) = \perp \). The product of sets (flat cpo’s) \( A \) and \( B \) is \( A \times B \) with pairing \((a,b)\) and projections \( \pi_1 \) and \( \pi_2 \).

3 mABS

We define a small concurrent object-based language mABS, short for milli-ABS, with asynchronous calls and futures, as depicted in fig. 1. The mABS language is an extension of the language \( \mu \)ABS (micro-ABS) of message-passing processes introduced in [11] with return values as futures, and it corresponds essentially to the asynchronous fragment of the core ABS language extensively studied in the EU FP7 project HATS. The language is fairly self-explanatory. A program is a sequence of class definitions, along with a set of global variables \( x \), and a "main" statement \( s \). The class hierarchy is flat and fixed. Objects have parameters \( x \), local variable declarations \( y \), and methods \( M \). Methods have parameters \( x \), local variable declarations \( y \) and a statement body. For simplicity we assume that variables have

| \( x, y \in \text{Var} \) | Variable |
| \( e \in \text{Exp} \) | Expression |
| \( P ::= CL(x, s) \) | Program |
| \( CL ::= class C(x)\{y, M\} \) | Class definition |
| \( M ::= m(x)\{y, s\} \) | Method definition |
| \( s ::= s_1; s_2 \mid x = rhs \mid \text{skip} \mid \text{if } e(s_1) \text{ else } s_2 \mid \text{while } e(s) \mid \text{return } e \) | Statement |
| \( rhs ::= e \mid \text{new } C(e) \mid e.m(e) \mid e.get \) | Right hand sides |

Figure 1: mABS abstract syntax
```java
class Server1(){
    serve(x){
        return foo(x)
    }
}
```

Figure 2: µABS Code Sample 1

```java
class Server(){
    serve(x){
        s1,s2,f1,f2,r1,r2,
        if small(x){
            return foo(x)
        }
        else {
            s1 = new Server() ;
            s2 = new Server() ;
            f1 = s1!serve(upper(x)) ;
            f2 = s2!serve(lower(x)) ;
            r1 = f1.get ;
            r2 = f2.get ;
            return combine(r1,r2)
        }
    }
}
```

Figure 3: µABS Code Sample 2

unique declarations. Expression syntax is left open, but is assumed to include the constant `self`. We require that expressions are side effect free. We omit types from the presentation. Types could be added, but they would not affect the results of the paper in any significant way and consequently left out.

Statements include standard sequential control structures, and a minimal set of constructs for asynchronous method invocation, object creation, and retrieval of futures (`get` statements).

**Example 3.1.** A very simple server applying `foo` to its argument is shown in fig. 2.

**Example 3.2.** Assume that `combine(upper(x),lower(x)) = foo(x)`. The program example in fig. 3 returns immediately with the result, if the argument to `serve` is small. If the argument is not small, two new servers are spawned, and computation of the result on upper and lower tranches is delegated to those servers. The results are then fetched from the two newly spawned servers by evaluating the `get` statements, combined, and returned.

## 4 Reduction Semantics

We first present an abstract “reference” semantics for mABS using rewriting logic. The presentation follows [11] quite closely. We use the abstract
semantics as the point of reference for the concrete network-oriented semantics which we present later. The reduction semantics uses a reduction relation $cn \rightarrow cn'$ where $cn$, $cn'$ are configurations, as determined by the runtime syntax in fig. 4. Later on, we introduce different configurations and transition relations, and so use index 1, or talk of e.g. configurations of "type 1", for this first semantics when we need disambiguate. Terms

| $x \in Var$ | Variables |
| $o \in OID$ | Object id |
| $p \in PVal$ | Primitive values |
| $f \in Fut$ | Futures |
| $v \in Val$ | $PVal \cup OID \cup Fut$ |
| $z \in Name$ | Names |
| $l \in MEnv$ | $Var \cup \{\text{ret}\} \rightarrow Val_\bot$ |
| $a \in OEnv$ | Object environment |
| $tsk \in Tsk$ | Task |
| $obj \in Obj$ | Object |
| $fut \in fut$ | Future |
| $call \in Call$ | Call |
| $ct \in Ct$ | Container |
| $cn \in Cn$ | Configuration |

Figure 4: mABS type 1 runtime syntax

of the runtime syntax are ranged over by $M$, and $\prec$ is the subterm relation. The runtime syntax uses disjoint, denumerable sets of object identifiers $o \in OID$, futures $f \in Fut$, and primitive values $p \in PVal$. Values are either primitive values, OID’s, or futures. Lifted values are ranged over by $v_\bot \in Val_\bot$, and we use $\sqsubseteq$ for the standard, associated partial ordering. OID’s and futures are subject to $\pi$-calculus like binding. Later, in the type 2 semantics, this type of explicit binding is dropped. Accordingly, names are either OID’s or futures, we use $z$ as a generic name variable, and names are bound using the $\pi$-like binder $\text{bind}$.

Standard alpha-congruence applies to name binding.

Configurations are "$\pi$-scoped" multisets of containers of which there are four types, namely tasks, objects, futures, and calls. Configuration juxtaposition is assumed to be commutative and associative with unit 0. In addition we assume the standard structural identities $\text{bind}\ z.0 = 0$ and $\text{bind}\ z.(cn_1 cn_2) = (\text{bind}\ z.cn_1) cn_2$ when $z \not\in \text{fn}(cn_2)$. We often use a vectorized notation $\text{bind}\ z.cn$ as abbreviation, letting $\text{bind}\ v.cn = cn$ where $v$ is the empty sequence. The structural identities then allows us to rewrite each configuration into a standard form $\text{bind}\ z.cn$ such that each member of $z$ occurs free in $cn$, and $cn$ has no occurrences of the binding operator $\text{bind}$. We use standard forms frequently.

Tasks are used for method body elaboration, and futures are used as centralized stores for assignments to future variables. Task and object envi-
environments \( l \) and \( a \), respectively, map local variables to values. Task environments are aware of a special variable \( \text{ret} \) that the task can use in order to identify its return future. Upon invocation, the task environment is initialized using the operation \( \text{locals}(o, f, m, v) \) by mapping the formal parameters of \( m \) in \( o \) to the corresponding actual parameters in \( v \), by initializing the method local variables to suitable null values, by mapping \( \text{self} \) to \( o \), and by mapping \( \text{ret} \) to \( f \), intended as the return future of the task being created. Object environments are initialized using the operation \( \text{init}(C, v) \), which maps the parameters of \( C \) to \( v \), and initializes the object local variables as above.

In addition to \( \text{locals} \) and \( \text{init} \), the reduction rules presented below use the following helper functions:

- \( \text{body}(o, m) \) retrieves the statement of the shape \( s \) in the definition body for \( m \) in the class of \( o \).
- \( \hat{e}(a, l) \in \text{Val} \) evaluates \( e \) using method environment \( l \) and object environment \( o \).

Calls play a special role in defining the external observations of a configuration \( cn \). An observation, or \( \text{barb} \), is a call expression of the form \( o!m(v) \), ranged over by \( \text{obs} \). In order to define the observations of a given configuration, we assume a fixed set \( \text{Ext} \) of external OID’s to which outgoing method calls can be directed. Names in \( \text{Ext} \) are not allowed to be bound. A \( \text{barb} \), then, is an external method call, i.e. a method call to an OID in \( \text{Ext} \). Calls that are not external are meant to be completed in usual reduction semantics style, by internal reaction with the called object, to spawn a new task. External calls could be represented directly, without introducing a special container type (which is not present in the core ABS semantics of [23]), by saying that a configuration \( cn \) has \( \text{barb} \, \text{obs} = o!m(v) \) if and only if \( cn \) has the shape

\[
\text{bind } o_1. \,(cn' o(a_2, a) \, t(o_2, l, e_1!m(e_2); s)),
\]

where \( e_1(a, l) = o \in \text{Ext} \) and \( e_2(a, l) = v \). However, in a semantics with unordered communication, which is what we are after, consecutive calls should commute, i.e. there should be no observational distinction between the method bodies \( e_1!m_1(e'_1); e_2!m_2(e'_2) \) and \( e_2!m_2(e'_2); e_1!m_1(e'_1) \). This, however, is difficult to reconcile with the representation (1). To this end call containers are introduced, to allow configurations like (1) to produce a corresponding call, and then proceed to elaborate \( s \).

We next present the reduction rules. For ease of notation the rules assume that sequential composition is associative with unit \( \text{skip} \). Figures 5 and 6 present the reduction rules. The rules use the notation \( cn \vdash cn' \rightarrow cn'' \) as shorthand for \( cn \, cn' \rightarrow \, cn \, cn'' \). We use \( \rightarrow_1 \) when we want to make the reference to the type 1 reduction semantics explicit. Fig. 5 gives the mostly routine rules for assignment, control structures, and contextual reasoning, and fig. 6 gives the more interesting rules that involve inter-object communication, external method invocation, and object creation. A method call causes a new future to be created, along with its future container, initialized to \( \bot \). Internal and external calls are treated somewhat asymmetrically.
If $cn_1 \rightarrow cn_2$ then $cn \vdash cn_1 \rightarrow cn_2$

Figure 5: mABS reduction semantics part 1

Since external calls are only used to define bars, and bars corresponding to the act of receiving a external return value are not very meaningful in a reduction semantics setting\(^2\). Future instantiation is done when return statements are evaluated, and get statements cause the evaluating task to hang until the value associated to the future is defined. Wait statements can easily be added; they contribute nothing essential to this presentation. Object creation (new) statements cause new objects to be created along with their OID’s in the expected manner.

We note some basic properties of the reduction semantics.

**Proposition 4.1.**

1. If $cn \rightarrow cn'$ then $fn(cn') \subseteq fn(cn')$

2. If $o(a, a') \preceq cn$, then $o(a, a') \preceq cn'$ for some object environment $a'$

3. If $f(f, v) \preceq cn$ then $f(f, v') \preceq cn'$ for some $v' \subseteq v'$ such that $v \subseteq v'\subseteq

\(^2\)They are very meaningful in labelled semantics setting, but that is a different story.

Figure 6: mABS reduction semantics part 2
Proof. No structural identity nor any reduction rule allows an OID or future to escape its binder. Also no rules allow futures to be re-instantiated. The result follows.

Consider a program \(CL\{x, s\}\). Assume a reserved OID \(main\) and a reserved future \(f_{init}\). A type 1 initial configuration is any configuration of the shape

\[cn_{init} = \text{bind} \, main, f_{init}, o(main, \bot) \, t(main, l_{init}, s) \, f(f_{init}, \bot)\]

where \(l_{init}\) is the initial type 1 method environment assigning suitable default values to the variables in \(x\), and \(l(ret) = f_{init}\). The program can place calls to any of the external objects with OID’s in \(Ext\), and in this way produce externally observable output.

We say that a configuration \(cn\) of type 1 is reachable if there is a derivation \(cn_{init} = cn_0 \rightarrow \cdots \rightarrow cn\) where \(cn_{init}\) is an initial configuration. Reachable configurations satisfy some well-formedness conditions which we make significant use of later in the paper.

**Definition 4.2 (Type 1 Well-formedness).** A configuration \(cn\) is type 1 well-formed (WF1) if \(cn\) satisfies:

1. **OID Uniqueness**: Suppose \(o(o_1, a_1), o(o_2, a_2) \preceq cn\) are distinct object occurrences. Then \(a_1 \neq a_2\)

2. **Task-Object Existence**: If \(t(o, l, s) \preceq cn\) then \(o(o, a) \preceq cn\) for some object environment \(a\)

3. **Object Existence**: Suppose \(o \notin Ext\) occurs in \(cn\). Then \(o(o, a) \preceq cn\) for some object environment \(a\)

4. **Object Nonexistence**: Suppose \(o \in Ext\). Then \(o(o, a) \not\preceq cn\) for any object environment \(a\)

5. **Object Binding**: Suppose \(o \notin Ext\). Then \(o \notin fn(cn)\)

6. **Future Uniqueness**: Suppose \(f(f_1, v_{\bot, 1}), f(f_2, v_{\bot, 2}) \preceq cn\) are distinct future occurrences. Then \(f_1 \neq f_2\)

7. **Single Writer**: If \(t(o, l, s) \preceq cn\) then \(f(l(ret), \bot) \preceq cn\)

8. **External Calls**: If \(c(o, o', m, v) \preceq cn\) then \(o' \in Ext\)

Well-formedness is important as it ensures that objects and futures, if defined, are defined uniquely, and that, e.g., tasks are defined only along with their accompanying object. The existence properties are needed to ensure that the partitioning of OID’s into external and (by extension) internal is meaningful, in that external references are always routed outside the “current configuration”. The Single Writer property reflects the fact that only the task that was spawned along with some given future is able to assign to that future, and hence, if the task has not yet returned, the future remain uninstantiated.
Proposition 4.3 (WF1 Preservation). If \( cn \) is WF1 and \( cn \rightarrow cn' \) then \( cn' \) is WF1.


Theorem 4.4. If \( cn \) is type 1 reachable then \( cn \) is WF1.

Proof. It is sufficient to check that any initial configuration is WF1, and then use proposition 4.3. □

5 Type 1 Barbed Equivalence

Our approach to implementation correctness uses barbed equivalence [31]. The goal is to show that it is possible to remain strongly faithful to the reference semantics, provided all nondeterminism is deferred to be handled by a separate scheduler. This allows to draw strong conclusions also in the case a scheduler is added, as we discuss in section 11. Barbed equivalence requires of a pair of equivalent configurations that the internal transition relation \( \rightarrow \) is preserved in both directions, while preserving also a set of external observations. Although weaker than corresponding equivalences such as bisimulation equivalence on labelled transition systems, barbed equivalence in nonetheless of interest for the following two reasons:

1. Barbed equivalence offers a reasonable notion of observationally identical behaviour on closed systems, i.e. when composition of (in our case) subconfigurations to build larger configurations is not considered because it a) is for some reason not important or relevant, or b) does not offer new observational capabilities.

2. Barbed equivalence can be strengthened in a natural way to contextual equivalence [29] by adding to barbed equivalence a natural requirement of closure under context composition. Furthermore, a number of works, cf. [22, 32] have established very strong relations between contextual equivalence for reduction oriented semantics and bisimulation (logical relation) based equivalences for sequential and higher-order computational models.

It is, however, far from trivial to devise a natural notion of context that works at the level of the network semantics introduced later, and such that the notions of context correspond at both the abstract, reference semantics level we consider at present, and at the network level. For this reason the account of this paper based on barbed equivalence is also a natural stepping stone towards a deeper study of the notion of context in real-world—or at least not overly artificial—networked software systems.

Let \( obs = o!hn(v) \). The observation predicate \( cn \downarrow obs \) is defined to hold just in case \( cn \) can be written in the form

\[
\text{bind } o.(cn' \ c(o, o', m, v)) .
\]

The derived predicate \( cn \downarrow obs \) holds just in case \( cn \rightarrow^* cn' \downarrow obs \) for some \( cn' \).
**Definition 5.1** (Type 1 Witness Relation, Type 1 Barbed Equivalence). Let $R$ range over binary relations on WF1 configurations. The relation $R$ is a **type 1 witness relation**, if $cn_1 R cn_2$ implies

1. $cn_2 R cn_1$ (symmetry)
2. If $cn_1 \rightarrow cn'_1$ then $cn_2 \rightarrow^* cn'_2$ for some $cn'_2$ such that $cn'_1 R cn'_2$ (reduction closure)
3. If $cn_1 \downarrow obs$ then $cn_2 \downarrow obs$ (barb preservation)

The WF1 configurations $cn_1$ and $cn_2$ are **type 1 barbed equivalent**, $cn_1 \cong cn_2$, if $cn_1 R cn_2$ for some type 1 witness relation $R$.

We establish some well-known, elementary properties of barbed equivalence for later reference.

**Proposition 5.2.** The identity relation is a type 1 witness relation. Barbed equivalence is a type 1 witness relation. If $R$, $R_1$, $R_2$ are type 1 witness relations then so is

1. $R^{-1}$
2. $R^*$
3. $R_1 \circ R_2 \circ R_1$

**Proof.** See appendix 1.

We may conclude that $\cong$ has the expected basic property:

**Proposition 5.3.** $\cong$ is an equivalence relation.

**Proof.** The result follows from prop. 5.2. For transitivity, in particular, we use prop. 5.2.3.

The following property of barbed equivalence illustrates well its closed system nature:

**Proposition 5.4.** Suppose $cn$ is WF1, $o \notin fn(cn)$, and $cn o(o,a)$ is WF1. Then $cn \equiv cn o(o,a)$

**Proof.** See appendix 1.

This is reasonable from a closed system perspective, as if $o \notin fn(cn)$ there is no way $o(o,a)$ can be exercised without outside stimulus, but barbed equivalence lacks a context formation clause that can allow such stimulus to be produced. Thus, from a modeling perspective, all stimulus (= model input) that needs to be considered in a given modeling exercise must be included from the outset.
6 Network Semantics

We now turn to the second, main part of the paper where we address the problem of efficiently executing mABS programs on an abstract network graph using the location independent routing scheme alluded to in the introduction. The approach follows closely the network semantics introduced in [11], with the important difference that return values, as futures, are now included. In addition to the naming, routing, and object migration issues already addressed in [11] the additional challenge is to ensure that futures are correctly assigned and propagated at the network level.

In the network semantics we assume an explicitly given network “underlay”: A network of nodes and directional links to which message buffers are associated, modeling a concrete network structure with asynchronous point-to-point message passing. Object execution is localized to each node. At the outset nodes know only of their “own” objects, but as routing information is propagated, inter node messaging becomes possible. Objects may choose to migrate between neighbouring nodes. When this is done is not addressed here; we discuss possible adaptation/scheduling strategies in a separate paper [12]. The propagation of routing information will automatically see to it that routing tables are eventually updated. How and when this is done is again left to a scheduler. Method calls can be issued if the caller task knows the OID of the called object. The call is delivered once a route to the callee is known.

For the language considered in this paper the network semantics must be extended to cover also return values and futures. In this paper we use a form of eager forward chaining [6]. Each object mentioning a future can subscribe to that future at some other object. This may happen in remote method calls where the caller subscribes to the return value later to be provided by the callee. It may also happen when a value containing a future is passed from some sender object to some receiver object. In that case the receiver object becomes subscriber at the sender object for that future. When a future gets instantiated to an actual value at some object, it is the task of that object to forward the instantiation to the subscribing objects. This is the implementation strategy applied in our work as well, and it is the objective of the proof to show that this approach is sound and fully abstract for our network semantics, even when routing is in an unstable state.

Example 6.1. An example illustrating future propagation and the interaction with routing is shown in Figure 7. In configuration 1 an rpc with argument \( f \), a future, is sent from object \( \text{obj}_{1} \) residing on \( \text{nd}_{1} \) is sent to object \( \text{obj}_{3} \) on node \( \text{nd}_{3} \). This causes the following events:

- The forwarding list for \( f \) at object \( \text{obj}_{1} \) is augmented to include object \( \text{obj}_{3} \)
- A new future \( g \) is created at object \( \text{obj}_{1} \), to hold the return value of method \( \text{obj}_{3.m} \)
- Object \( \text{obj}_{3} \) is augmented with placeholders for \( f \) and \( g \), and the forwarding list for \( g \) is augmented to point to \( \text{obj}_{1} \)
The scheduler now decides to migrate \textit{obj}_3 from \textit{nd}_3 to \textit{nd}_2. No action is required other than regular routing table updates, as the forward pointers keep pointing to the same objects. Finally, the remote method returns the future \textit{f}. The forwarding chain for \textit{g} has now outlived its purpose and can be garbage collected (our semantics does not actually accomplish this quite yet). The forwarding chain for \textit{f} is augmented to point to the return future \textit{g}. The resulting loop costs additional messaging in the current rather basic semantics, and could be eliminated. However, it does not cause additional latency as \textit{obj}_1 is able to discover an assignment to \textit{f} at \textit{obj}_1 when it is first made.

6.1 mABS-NET Runtime Syntax

In fig. 8 we present the mABS-NET runtime syntax, i.e. the shape of the runtime state. Recall from section 4 that we reuse symbols as much as possible and use indices to disambiguate. Thus, for instance, \textit{Obj}_1 is the set \textit{Obj} of the type 1 semantics in fig. 4, and \textit{Obj}_2 is the corresponding set in fig. 8. We adopt the same syntactical conventions as in section 4. Tasks are

\footnote{This might not appear wholly optimal, but for the sake of the example...}
 unchanged from fig. 4. We write t(cn) for the multiset of tasks in cn, i.e. the multiset \{tsk | \exists cn’.cn = tsk cn’\}, and o(cn) for the multiset of objects in cn, similarly defined. We also write m(cn) for the multiset \{msg | msg ⪯ cn\}.

We proceed to explain the different types of containers and the operations on them, concentrating on the treatment of futures. For a detailed explanation of other features, in particular routing, we refer to [11].

**Network and Routing** The network graph contains a vertex \(u\) for each node container \(n(u, t)\) and an edge \((u, u')\) for each link \(l(u, q, u')\). The reduction semantics given later does not allow nodes or links to be dynamically changed, so in the context of any given transition (or, execution), the network graph remains constant. Note that there is no a priori guarantee that the network graph is a well-formed graph. For the remainder of the paper we impose some constraints on the well-formedness of the network graph, including (i) endpoints of edges exist, (ii) vertices and edges are uniquely determined, (iii) the network graph is reflexive and symmetric, and (iv) the network graph is connected. For routing we adopt a simple Bellman-Ford distance vector (d.v.) discipline. For a routing table \(t\), \(t(o) = (u, n)\) indicates that, as far as \(t\) is concerned, there is a path from the current node (the node to which \(t\) is attached) to the node \(u\) with distance \(n\) that first visits the node \(u\). We only count hops in this work, for simplicity. A more realistic routing scheme attaches weights to the edges, reflecting latency or capacity constraints. Next hop lookup is performed by the operation \(\text{nxt}(o, t) = \pi_1(t(o))\) where \(\pi_1\) is the first projection. There is also an operation of updating a routing table \(t\) by a routing table \(t'\) received from a neighbouring node \(u\),
defined by the function

\[
\text{upd}(t, u, t')(o) = \begin{cases} \\
\bot & \text{if } o \not\in \text{dom}(t) \cup \text{dom}(t') \\
t(o) & \text{else, if } o \not\in \text{dom}(t') \\
(u, \pi_2(t'(o)) + 1) & \text{else, if } \pi_1(t'(o)) = u \\
(u, \pi_2(t'(o)) + 1) & \text{else, if } t'(o) < \pi_2(t(o)) - 1 \\
t(o) & \text{otherwise}
\end{cases}
\]

Finally, there is an operation \(\text{reg}(o, u, t)\) that returns the routing table \(t'\) obtained by registering \(o\) at \(t's\) current node \(u\), i.e. such that

\[
\text{reg}(o, u, t)(o') = \begin{cases} \\
(u, 0) & \text{if } o = o' \\
t(o') & \text{otherwise}
\end{cases}
\]

**Message Queues** Queue operations are standard: \(\text{enq}(v, q)\) enqueues \(v\) onto the tail of \(q\), \(\text{hd}(q)\) returns the head of \(q\), and \(\text{deq}(q)\) returns the tail of the \(q\), i.e. \(q\) with \(\text{hd}(q)\) removed. If \(q\) is empty then \(\text{hd}(q) = \text{deq}(q) = \bot\).

**Objects and Object Environments** Objects \(o(a, u, q_{in}, q_{out})\) are now attached to a node \(a\) and a pair of an ingoing \((q_{in})\) and an outgoing \((q_{out})\) fifo message queue, and the notion of object environment is refined to take futures into account in a localized manner. In the type 2 semantics, object environments \(a\) are now augmented by mapping futures \(fut\) to pairs \((v_\bot, o)\) where:

- \(v_\bot\) is the lifted value currently assigned to \(fut\) at the current object, and

- \(o\) is a forwarding set of the objects subscribing to updates to \(fut\) at the current object.

For instance, if \(a(fut) = (\bot, o_1 :: o_2 :: \varepsilon)\) the future \(fut\) is as yet uninstantiated (at the object to which \(a\) belongs), and, if \(fut\) eventually does become instantiated, the instantiation must be forwarded to \(o_1\) and \(o_2\), in random order.

We introduce some syntax to help manipulating object environments:

- \(a(x)\) abbreviates \(\pi_1(a)(x)\), \(a(f)\) abbreviates \(\pi_2(a)(f)\)

- \(a[v/x]\) is \(a\) with \(\pi_1(a)\) replaced by the expected update. Similarly \(a[v/f]\) updates \(\pi_2(a)\) by mapping \(f\) to the pair \((v, \pi_2(a(f)))\), i.e. the assigned value is updated and the forwarding list remains unchanged. If \(f \not\in \text{dom}(\pi_2(a))\) then \(a[v/f](f) = (v, \varepsilon)\), i.e. the update to value takes effect. Finally we use \(a[v, o]/f\) for the expected update where both the value and the forwarding list is updated.

- \(fw(v, o, a)\) updates \(\pi_2(a)\) by for each future \(f\) occurring in \(v\) adding \(o\) to the forwarding list of \(a(f)\), i.e. by mapping \(f\) to the pair either \((\bot, o)\) if \(a(f)\) is undefined \((= \bot)\), or \((\pi_1(a(f)), o :: \pi_2(a(f)))\) otherwise.
• $\text{init}(C, v)$ returns an initial objects environment by mapping the formal parameters of $C$ to $\rightarrow$.
• $\text{init}(f, a)$ augments $a$ by mapping $f$ to the pair $(\bot, \varepsilon)$. If $f \not\in \text{dom}(a)$ then $\text{init}(f, a) = a$.
• $\text{init}(v, a)$ augments $a$ by mapping each $f$ in $v$ which is uninitialized in $a$ (i.e. such that $f \not\in \text{dom}(a)$) to $(\bot, \varepsilon)$.

As a consequence of this change, futures are eliminated as containers in the type 2 runtime syntax. In other respects, the type 2 runtime syntax is unchanged: Syntactical conventions that are not explicitly modified in the type 2 syntax above are unchanged, in particular we continue to assume multiset properties of configuration juxtaposition.

### Messages

The network semantics uses four types of messages. The first is a method call message of the shape $\text{call}(o, o', f, m, v)$, already implicit in the reduction semantics (and explicit for external calls). The remaining three are new: $\text{future}(o, f, v)$ is a future instantiation message, informing object $o$ that $f$ now has been instantiated to value $v$, $\text{table}(t)$ encodes the routing table $t$, and $\text{object}(cn)$ is used for object migration. The first two types of messages are said to be object bound, and the two latter are node bound. We define $\text{dst}(msg)$, the destination of $msg$ to be $o'$ for $msg$ of the first form above, $o$ for $msg$ of the second form, and $\text{dst}(msg) = \bot$ in the remaining two cases. For an object message $\text{object}(cn)$ to be valid, the configuration $cn$ needs to be an object closure of the form $o(o', a', u', q'\text{in}, q'\text{out}) t(o, l_1, s_1) \ldots tsk(o, l_n, s_n)$. Specifically, if $cn$ is any configuration then $\text{clo}(cn, o)$, the closure of object $o$ with respect to $cn$, is the multiset of all type 2 containers of the form either $o(o', a', u', q'\text{in}, q'\text{out})$ or $t(o', l', s')$ such that $o' = o$, and $\text{objof}(cn)$ is a partial function returning $o$ if all type 2 containers in $cn$ are either objects or tasks, with OID $o$.

### 6.2 Reduction Semantics

An important distinction between the reference semantics and the network semantics is the absence of binding. For the standard semantics, name binding plays an important role to avoid clashes between locally generated names. However, in a language with BID’s this device is no longer needed, as globally unique name can be guaranteed easily by augmenting names with their generating NID. Since all name generation in the $\mu$ABS-NET semantics below takes place in the context of a given NID, we can simply assume operations $\text{newf}(u)$, resp. $\text{newo}(u)$, that return a new future, resp. OID, which is globally fresh for the “current context”. We use $\text{new}(z)$ for either $\text{newf}$ or $\text{newo}$ when the nature of $z$ is not known.

We present the mABS-NET reduction rules. First, fig. 5 applies with the following two minor modifications:

• Rule ctxt-2 is dropped as name binding is dropped from the type 2 runtime syntax.
• Rule wfield is modified in the obvious way to read: If $x \in \text{dom}(a)$ then
$\alpha(o, a, u, q_{in}, q_{out}) \triangleright t(o, l, x = e; s) \rightarrow \alpha(o, a[e(a, l)/x], u, q_{in}, q_{out}) \triangleright t(o, l, s)$

The remaining reduction rules are presented in fig. 9.

---

**Routing**  The first set of rewrite rules, t-send and t-rcv, are concerned with the exchange of routing tables.
Message Passing The three rules msg-send, msg-rcv and msg-route are used to manage message passing, i.e. reading a message from a link queue and transferring it to the appropriate object in-queue, and dually, reading a message from an out-queue and transferring it to the attached link queue. If the destination object does not reside at the current node, the message is routed to the next link. In rule msg-rcv note that the receiving node is not required to be present. This, however, will be enforced by the well-formedness condition later, which prohibits output links.

Unstable Routing The two rules msg-delay-1 and msg-delay-2 are used to handle the case where routing tables have not yet stabilized. For instance it may happen that updates to the routing tables have not yet caught up with object migration. In this case, a message may enter an out-queue without the hosting nodes routing table having information about the message’s destination (rule msg-delay-2). Another case is where a node receives a message on a link without knowing where to forward it (rule msg-delay-1). This situation is particularly problematic as a blocked message may prevent routing table updates to reach the hosting node, thus causing deadlock. The solution we propose is to use a network self-loop as a buffer for temporarily unroutable messages.

Producing and Consuming Messages The four rules call-send, call-rcv, fut-send, fut-rcv produce and consume messages, method calls and future instantiations. A method call causes a local future to be created and passed with the call message. Upon reception of the call, the callee first initialized those received futures it does not already know about, and then augments the resulting local object environment to forward instantiations of the received future to the caller. Observe that it may be that the callee already knows about the return future of the call. Since message order is not assumed to be preserved a later call referring to the original return future may overtake the earlier call. The eventual return value becomes bound to the return future by the assignment to the constant ret during initialization of the called methods local environment. The rule fut-send may cause future instantiations to be forwarded to objects in the forwarding list whenever the future is seen to have received a value, and fut-rcv causes the receiving object to update its local environment accordingly. A future may itself be instantiated to a future. The local forwarding table may thus need to be updated.

Language Constructs The three rules ret-2, get-2, new-2 handle the corresponding language constructs. Return statements cause the corresponding future to be instantiated, as explained. Get statements read the value of the future provided it has received a value, and new statements cause a new object to be created, initialized, and registered at the local node.

Object Migration The final rules concern object migration. Those rules are global in that they are not allowed to be used in subsequent applications
of the ctxt-1 rule. In this way we can guarantee that only complete object closures are migrated. In rule obj-send, \( cn - cn' \) is multiset difference.

It is important to notice that all of the above rules are strictly local and appear only to mechanisms directly implementable at link level: Tests and simple datatype manipulations taking place at a single node, or accesses to a single nodes link layer interface. The “global” property appealed to above for the migration rules is merely a formal device to enable an elegant treatment of object closures.

The reduction rules can be optimized in several ways. For instance, object self-calls are always routed through the “network interface”, i.e. the hosting nodes self-loop. This is not necessary. It would be possible to add a rule to directly spawn a handling task from a self call without affecting the results of the paper. We note some elementary properties of the type 2 semantics.

**Proposition 6.2.** Suppose that \( cn \rightarrow cn' \).

1. If \( n(u, t) \leq cn \) then \( n(u, t') \leq cn' \) for some \( t' \).
2. If \( lnk = l(u, q, u') \leq cn \) then \( l(u, q', u') \leq cn' \) for some \( q' \).
3. If \( obj = o(a, u, \{ q_{in}, q_{out} \}) \leq cn \) then there is an object
   \[ obj' = o(a', u', \{ q'_{in}, q'_{out} \}) \leq cn' \]
   (the derivative of \( obj \) in \( cn' \)) such that \( a' = a \), \( u' = u \), for all \( x \), if \( a'(x) \downarrow \) then \( a'(x) \downarrow \), for all \( f \), if \( a'(f) \downarrow \) then \( a'(f) \downarrow \), and if \( \pi_1(a(f)) \downarrow \) then \( \pi_1(a'(f)) \downarrow \).
4. If \( tsk = t(o, l, s) \leq cn \) then either there is a task \( t(o', l', s') \leq cn' \) (the derivative of \( tsk \) in \( cn \)) such that \( o' = o \), \( dom(l) \subseteq dom(l') \), and \( l'(\text{ret}) = l(\text{ret}) \), or else there is an object \( o(a, u, \{ q_{in}, q_{out} \}) \) such that \( \pi_1(a(l(\text{ret}))) \downarrow \).

**Proof.** By inspecting the rules.

We then turn to initial configurations. Let a program \( CL\{x, s\} \) be given with reserved OID \( \text{main} \) and a reserved future \( f_{init} \).

**Definition 6.3** (Type 2 Initial Configuration). A type 2 initial configuration has the shape

\[
\text{cn}^{init} = \text{cn}^{graph} \circ (\text{main}, a^{init,2}, u^{init}, \varepsilon, \varepsilon) \circ t(\text{main}, l^{init,2}, s)
\]

where:

- \( l^{init,2} \) identical to \( l^{init,1} \), except that \( l^{init,2}(f_{init}) = \bot \),
- \( a^{init,2} = \bot[\bot, \varepsilon]/f_{init} \),
- \( \text{cn}^{graph} \) is a configuration consisting only of nodes and links,
- \( u^{init} \) names a node \( n(u^{init}, l^{init}) \) in \( \text{cn}^{graph} \),
- \( t^{init}(o) = (u^{init}, 0) \), and \( t^{init}(o') = \bot \) for \( o' \neq o \), and
- \( t(o) = \bot \) for all routing tables \( t \neq t^{init} \) in \( \text{cn}^{init} \).

20
7 Type 2 Well-formedness

Well-formedness becomes more complex in the case of the network semantics, as account must be taken of e.g. queues, messages in transit, and routing, to ensure that e.g., multiple nodes are never given identical names, that futures are never assigned inconsistent values, as detailed below. A particularly delicate matter concerns the way future instantiations are propagated. The well-formedness condition needs to ensure that either all objects that may some time need the value of a future will also eventually receive it, or else no object does so (in case the method for which the future holds the return value fails to terminate). This is the “future liveness” property in def. 7.4 below. To this end we first define when a future $f$ is active at a given object.

Definition 7.1 (Active Futures). Fix a configuration $cn$. The future $f$ is active in $o$ if one of the following two conditions hold:

1. There is an object container $o(a, u, q_{in}, q_{out}) \preceq cn$ such that $a(f) \downarrow$.
2. A message call($o', o, f', m, v$) is in transit in $cn$ and $f$ occurs in $v$.

Thus a future $f$ is active in $o$ if either $o$ already has a value for $f$, or $o$ has received $f$ but not its value, or if $o$ is about to receive a method call which contains $f$ among one of its parameters.

We then define which objects are due to be notified by eventual future instantiations.

Definition 7.2 (Notification Path). Fix a type 2 configuration $cn$ and an object container $o(a, u, q_{in}, q_{out}) \preceq cn$. Let $n \geq 1$. Inductively, $o$ is on the notification path of $f$ in $n$ steps, if $n$ is minimal such that one of the following conditions hold:

1. $n = 0$, and $\pi_1(a(f)) = v \neq 0$
2. $n = 1$, and there is a task $t(o, l, s) \preceq cn$ with $l(ret) = f$
3. $n = 1$, and there is a future message future($o, f, v$) $\preceq cn$
4. $n = n' + 1$ and there is an object $o(o', u', q'_{in}, q'_{out}) \preceq cn$ such that $o \in \pi_2(a'(f))$, and $o'$ is on the notification path of $f$ in $n'$ steps
5. $n = 2$, and there is a call message call($o, o', f, m, v$) $\preceq cn$

Say that $o$ is on the notification list of $f$ if $o$ is on the notification list of $f$ in some number of steps.

Condition 1 is the base case when $f$ has already been instantiated. Condition 2 holds if $o$ is due to receive the return value from one of its pending tasks. Condition 3 holds if a future has been resolved and a future message is in transit to $o$. Condition 4 holds if $o$ has been inserted into a forwarding list for $f$ at closer distance to the “source”, and condition 5 holds if a call has been sent off to $o'$ with return future $f$. $o$ is then on the notification list.
of \( f \) since the call message is guaranteed to be received at the callee’s site and the forwarding list there updated. No clause corresponding to a future being transmitted from \( o' \) to \( o \) as part of the argument list, as in that case rule call-send sees to it that \( o' \)'s forwarding list is updated already when the call is made.

We prove that if \( o \) is on the notification path of \( f \) then in the next configuration \( o \) remains on the notification path of \( f \), without increasing the number of steps.

**Lemma 7.3.** Fix \( cn \) and an object \( \text{obj} = o(a, u, q_{in}, q_{out}) \preceq cn \). If \( o \) is on the notification path of \( f \) in \( n \) steps in the configuration \( cn \), \( cn \rightarrow cn' \), then \( o \) is on the notification path of \( f \) in at most \( n \) steps in \( cn' \).

**Proof.** See appendix 2.

We say that \( cn \) assigns \( v \) to \( f \) if there is an object container

\[
o(a, u, q_{in}, q_{out}) \preceq cn
\]

such that \( a(f) = v \neq \perp \), or else there is a future message \( \text{future}(o, f, v) \preceq cn \).

**Definition 7.4 (Type 2 Well-formedness).** A type 2 configuration \( cn \) is type 2 well-formed (WF2) if \( cn \) satisfies:

1. **OID Uniqueness:** If \( o(a, u, q_{in}, q_{out}), i \in \{1, 2\}, \) are distinct object occurrences in \( cn \) then \( o_1 \neq o_2 \)

2. **Object-Node Existence:** If \( o(a, u, q_{in}, q_{out}) \in cn \) then \( n(u, t) \in cn \) for some \( t \)

3. **Task-Object Existence:** If \( t(o, l, s) \preceq cn \) then \( o(a, u, q_{in}, q_{out}) \preceq cn \) for some \( a, u, q_{in}, q_{out} \)

4. **Object Existence:** If \( o \notin \text{Ext} \) occurs in \( cn \) then \( o(a, u, q_{in}, q_{out}) \preceq cn \) for some \( a, u, q_{in}, q_{out} \)

5. **Object Nonexistence:** Suppose \( o \in \text{Ext} \). Then \( o(a, u, q_{in}, q_{out}) \not\preceq cn \) for any \( a, u, q_{in}, q_{out} \)

6. **Buffer Cleanliness:** If \( o(a, u, q_{in}, q_{out}) \preceq cn \) and \( \text{msg} \preceq q_{in} \) or \( \text{msg} \preceq q_{out} \) then \( \text{msg} \) is object bound. Also, if \( \text{msg} \preceq q_{in} \) then \( \text{dst}(\text{msg}) = o \)

7. **Local Routing Consistency, 1:** If \( n(u, t), o(a, u, q_{in}, q_{out}) \in cn \) then \( \text{nxt}(o, t) = (u, 0) \)

8. **Local Routing Consistency, 2:** If \( n(u, t) \preceq cn \) and \( \pi_1(\text{nxt}(o, t)) = u' \) then there is a link \( l(u, q, u') \preceq cn \)

9. **Future Uniqueness:** If \( cn \) assigns \( v_i \) to \( f_i \), \( i \in \{1, 2\} \), then \( v_1 = v_2 \)

10. **Single Writer:** If \( t(o, l, s) \preceq cn \) then \( cn \) does not assign any \( v \) to \( l(\text{ret}) \) and no message \( \text{future}(o', l(\text{ret}), v) \) is in transit for any \( o', v \).
11. **Future Liveness**: If \(f\) is pending in \(o\) then \(o\) is on the notification path from \(f\)

Conditions 7.4.1 to 7.4.8 are inherited from [11]. Condition 7.4.11 is the future propagation property discussed above. We use the term “future liveness” not as a guarantee that \(f\) will eventually be instantiated, but to indicate that, if eventually \(f\) is instantiated somewhere a notification path exists along which the update can be propagated. The rationale behind 7.4.10 is that mABS enforces a single-writer discipline on futures which must be reflected in the well-formedness constraints at the mABS-NET level. Once the future has been assigned through the evaluation of a return statement, the task is “garbage collected”. For 7.4.6 observe that only object bound messages (for in-queues, with messages appropriately addressed) enter the object queues. Buffer cleanliness is needed to prevent the formation of contexts that are deadlocked because an in- or out-queue contains messages of the wrong type. For 7.4.7 the requirement should hold only when the object is not in transit, as otherwise the object may be on the wire away from node \(u\), and \(u\):s routing table will then have been updated.

**Lemma 7.5** (Type 2 Well-formedness Lemma). If \(cn\) is WF2 and \(cn \rightarrow cn'\) then \(cn'\) is WF2.

**Proof.** See appendix 2. \(\square\)

**Corollary 7.6.** If \(cn\) is type 2 reachable then \(cn\) is WF2.

**Proof.** First check that initial configurations are WF2 and closed and then use lemma 7.5. \(\square\)

An easy but important consequence of type 2 well-formedness is that assignments to futures cannot be updated.

**Proposition 7.7.** Suppose that \(cn\) is WF2. If \(cn\) assigns \(v\) to \(f\) and \(cn'\) assigns \(v'\) to \(f\) then \(v = v'\).

**Proof.** Since \(cn\) is WF2, if \(cn\) assigns \(v\) to \(f\) there cannot be a task \(t(o, l, s) \preceq cn\) such that \(l(ret) = f\). But the only way of assigning \(v' \neq v\) to \(f\) is through \(\text{ret-2}\), the result follows. \(\square\)

## 8 Type 2 Barbed Equivalence

We adapt the notion of barbed equivalence to the type 2 setting as in [11]. The only difficulty is to define the type 2 correlate of the observation predicate. Say an observation \(obs = \text{call}(o', o, m, v)\) is enabled at a configuration \(cn\) if a corresponding call message \(\text{call}(o', o, m, v)\) is located at the head of one of the object output queues in \(cn\). More precisely, the type 2 observability predicate is \(cn \downarrow obs\), holding if and only if \(cn\) has the following shape:

\[
\begin{align*}
\text{cn} &= \text{cn}' \ o(o', a, u, q_{\text{in}}, q_{\text{out}}) \\
\text{hd}(q_{\text{out}}) &= \text{call}(o', o, m, v)
\end{align*}
\] (2)

and \(\text{hd}(q_{\text{out}})\) is defined and equal to \(\text{call}(o', o, m, v)\).
It may be thought that the fifo queue discipline goes against the treatment of external calls in the type 1 semantics as there an external call container, once created, will remain as an inert element of all future configurations. Thus, once, say, 5 call containers have been created, all 5 calls can be observed at all configurations from that point onwards. This is obviously not the case in the type 2 semantics. On the other hand, in the type 2 semantics, external call messages can always be recycled on the reflexive link, allowing available external calls to be shuffled.

There are other ways of defining the observability predicate that may be more natural. For instance one may attach external OID’s to specific NID’s and restrict observations to those NID’s accordingly. It is also possible to add dedicated output channels to the model, and route external calls to those. None of these design choices have any effect on the subsequent results, however, but add significant notational overhead, particular in the latter case.

With the observation predicate set up, the weak observation predicate is derived as in section 5, and, as there, we define a type 2 witness relation as a relation that satisfies symmetry, reduction closure, and barb preservation. Thus:

**Definition 8.1.** Type 2 Barbed Equivalence Let $cn_1 \approx_2 cn_2$ if and only if $cn_1 \mathcal{R} cn_2$ for some type 2 witness relation $\mathcal{R}$.

In fact, for the purpose of this paper there in no real need to distinguish between the type 1 and type 2 equivalences, and hence we conflate the notions of witness relation and barbed equivalences, by letting the type of the configuration arguments be determined by the context, and use $\cong$ as the generic notion.

### 9 Normal Forms

An mABS-NET program can be run from an initial state in either the type 1 or the type 2 semantics. We want to show that the behaviour of the programs is preserved, in the sense that the initial states at type 1 and type 2 levels are barbed equivalent.

The key to the proof is a normal form lemma for mABS-NET saying, roughly, that any well-formed type 2 configuration can be rewritten into a form where queues have been emptied of all routable messages, where routing tables have been in some expected sense normalized, where all futures that are assigned a value somewhere are assigned a value everywhere the value might be needed (by well-formedness this value is unique), and where all objects have been moved to a single node. We perform this rewriting two steps:

- First we show that routing can be stabilized and inter node links emptied, except for external messages (messages addressed at an external OID). This part if the proof is identical to the corresponding proof in [11]. For this reason we only describe it cursorily here.
Algorithm 1: Stabilize routing and read internal link messages

**Input** Type 2 well-formed configuration $cn$ on a connected network graph

**Output** Configuration with stable routing and external link messages only

**repeat**
- Use $t$-send on each proper link in $cn$ to broadcast routing tables to all neighbours;

**repeat**
- Use $t$-rcv to dequeue one message on a link in $cn$ until $t$-rcv no longer enabled;
- Use $msg$-$rcv$, $msg$-$route$, $msg$-$delay$-$1$, $obj$-$rcv$ to dequeue one message from each link, if possible

until link queues contain only external messages, and routing is stable

Figure 10: Algorithm 1 – Stabilize routing and empty link queues of internal messages

- We then complete the construction by emptying object queues, propagating futures, and moving all objects to a single node.

## 9.1 Stabilization

In the context of a configuration $cn$ call a **proper link** any link $l(u,q,u')$ for which $u \neq u'$.

### Definition 9.1 (Stable Routing, External Queued Messages)

Let $cn$ be a type 2 configuration.

1. $cn$ has **stable routing**, if for all $nd(u,t)$, $o(o,a,u',q_{in},q_{out}) \preceq cn$, if $\text{nxt}(o,t) = u''$ then there is a minimal length path from $u$ to $u'$ which visits $u''$

2. $cn$ has **external link messages only**, if $l(u,q,u') \in cn$ and $msg \preceq q$ implies $msg$ is external.

The idea is to first empty link queues as far as possible as we simultaneously exchange routing tables to converge to a configuration with stable routing. This first stage is accomplished using algorithm 1 in fig. 10 where we hide uses of $\text{ctxt}$-$1$ to allow the transition rules to be applied to arbitrary containers. Write $A_1(cn) \leadsto cn'$ if $cn'$ is a possible result of applying algorithm 1 to $cn$. We then say that $cn'$ is in **stable form**. Stable forms are almost unique, but not quite, since routing may stabilize in different ways.

### Proposition 9.2

**Algorithm 1 terminates.**

**Proof.** See appendix 3. \hfill $\Box$

Let $t_1(cn) = \{tsk \mid tsk \preceq cn\}$ and let $o_1(cn)$ be the multiset of object containers $ct = o(o,a,u,q_{in},q_{out})$ in $cn$ such that either $ct \in o(cn)$, or else $o(o,a,u',q_{in},q_{out})$ is in transit in $cn$ from some $u'$ to $u$ (since then, after
applying alg. 1, u will host the object). Finally, let \( m_1(cn) \) be the multiset of external messages in transit in \( cn \), or of messages occurring in an object in- or out-queue.

**Proposition 9.3.** If \( A_1(cn) \leadsto cn' \) then

1. \( \text{graph}(cn) = \text{graph}(cn') \)
2. \( cn' \) has stable routing
3. \( cn' \) has external link messages only
4. \( t(cn') = t_1(cn) \)
5. \( o(cn') = o_1(cn) \)
6. \( m(cn') = m_1(cn) \)

**Proof.** Property 1 and 2 are immediate. Property 3 and 4 can be read out of the termination proof. For the remaining three properties observe first that \( t_1, o_1, \) and \( m_1 \) are all invariant under the transitions used in algorithm 1. The equations follows by noting that only external messages (and so no object closures) are in transit in \( cn' \).

Prop. 9.3 shows the “almost uniqueness” property alluded to above. The normal form property suggested by prop. 9.3 motivates a notion of equivalence “up to stabilization” defined below.

**Definition 9.4 \((\equiv_1)\).**

1. Let \( cn_1 R_1 cn_2 \) if \( cn_1 \) and \( cn_2 \) are WF2, \( \text{graph}(cn_1) = \text{graph}(cn_2) \), \( t_1(cn_1) = t_1(cn_2) \), \( o_1(cn_1) = o_1(cn_2) \), and \( m_1(cn_1) = m_1(cn_2) \).
2. Let \( cn_1 \equiv_1 cn_2 \) if there are \( cn'_1, cn'_2 \) such that
   \[ A_1(cn_1) \leadsto cn'_1 \leadsto cn'_2 \leadsto A_1(cn_2) \]

Prop. 9.3 together with termination of \( A_1 \) allows the existential quantifiers in def. 9.4.2 to be exchanged by universal ones.

**Corollary 9.5.** If \( A_1(cn) \leadsto cn' \) then \( cn \equiv_1 cn' \)

**Proof.** We have \( A_1(cn) \leadsto \leadsto A_1(cn') \).

**Lemma 9.6.** \( \equiv_1 \) is reduction closed

**Proof.** See appendix 3.

**Proposition 9.7.** \( \equiv_1 \) is a type 2 witness relation
Algorithm 2: Normalization

**Input** Type 2 well-formed configuration \( z : cn \) on a connected network graph

**Output** Configuration in type 2 normal form

- fix a NID \( u \);
- run alg. 1;
- while some object queue is nonempty {
  - use msg-send, msg-delay-2, call-rcv, fut-rcv to dequeue one message from each nonempty object queue;
  - while fut-send is enabled { apply fut-send };
- while an object exists not located at \( u \) {
  - use obj-send to send the object towards \( u \);
  - run alg. 1 }

Figure 11: Algorithm 2 – Normalization

**Proof.** Symmetry is immediate, and reduction closure follows by lemma 9.6. For barb preservation, if

\[
\begin{align*}
\text{cn}_1 &\equiv_1 \text{cn}_2 \\
\text{cn}_1 &\rightarrow^* \text{cn}'_1
\end{align*}
\]

and \( \text{cn}'_1 \downarrow \text{tick}(n) \) then by lemma 9.6 we find \( \text{cn}'_2 \) such that

\[
\begin{align*}
\text{cn}'_1 &\equiv_1 \text{cn}'_2
\end{align*}
\]

and by corollary 9.5 and transitivity of \( \equiv_1 \) we can assume that \( \text{cn}'_2 \) has external link messages only. But then \( \text{cn}'_2 \downarrow \text{tick}(n) \) as well. \( \square \)

**Corollary 9.8.** If \( A_1(cn) \rightsquigarrow cn' \) then \( cn \equiv cn' \)

**Proof.** By prop. 9.7 and corollary 9.5. \( \square \)

### 9.2 Normalization

We then turn to the second normalization step, to empty object queues, propagate futures, and migrate all object closures to a central node. The normalization procedure is algorithm 2 shown in fig. 11. Let \( A_2(cn) \rightsquigarrow cn' \) if \( cn' \) is a possible result of applying algorithm 2 to \( cn \). Initially a node \( u \) is chosen towards which all objects will migrate during normalization. Normalization is performed in cycles, each cycle starting and ending in a stable configuration. In each cycle one message is read from the object in- and out-queues. By well-formedness, object queues contain only calls and future messages. Receptions of future messages may cause object environment to instantiate futures. This may cause new future instantiation messages to be enabled. Accordingly, those messages are generated and sent to the objects out-queue. Once this is done, objects not yet at \( u \) will be migrated.

**Proposition 9.9.** Algorithm 2 terminates
Proof. See appendix 3.

We then turn to normal forms and define first a couple of ancillary operations. Let \( t_2(cn) \) be the multiset of method containers \( tsk = t(o, l, s) \) such that one of the following cases apply:

- \( tsk \) is a task container in \( cn \).
- There is a message call\((o', o, f, m, v)\) in transit, \( o \) is not external, \( l = \text{locals}(o, m, f, v) \) and \( s = \text{body}(o, m) \).

Let \( o_2(cn) \) be the multiset of object containers \( o(o, a', u', \epsilon, \epsilon) \) for which the following apply:

- \( u' = u \)
- There is an object container \( \text{obj} = o(o, a', u'', q_{in}, q_{out}) \preceq cn \)
- \( a'(x) = a(x) \) for all variables \( x \)
- \( a'(f) = (v, \epsilon) \) with \( v \neq \bot \) if and only if for some object container \( o(o_1, a_1, u_1, q_{in}, q_{out}) \preceq cn \), \( a_1(f) = (v, o) \), and otherwise \( a'(f) = (\bot, \epsilon) \), if for some such \( a_1, a_1(f) = (\bot, o) \).

Also say that \( cn \) has external messages only, if link queues in \( cn \) contain only external messages.

**Definition 9.10 (Normal Form).** A well-formed configuration \( cn \) is in normal form, if

1. \( cn \) has stable routing
2. \( cn \) has external messages only
3. \( t(cn) = t_2(cn) \)
4. \( o(cn) = o_2(cn) \)
5. \( m(cn) = m_1(cn) \)

**Proposition 9.11.** Suppose \( cn \) is WF2. If \( A_2(cn) \not\Rightarrow cn' \) then

1. \( cn' \) is in normal form
2. \( \text{graph}(cn) = \text{graph}(cn') \)
3. \( t_2(cn) = t(cn') \)
4. \( o_2(cn) = o(cn') \)
5. \( m_1(cn) = m(cn') \)

Proof. See appendix 3. \( \square \)

Proposition 9.11 motivates the following definition of normal form equivalence.
Definition 9.12 (≡2). 1. Let \( cn_1 \) \( \mathcal{R}_2 \) \( cn_2 \) if and only if \( cn_1 \) and \( cn_2 \) are WF2, graph(\( cn_1 \)) = graph(\( cn_2 \)), \( t_2(cn_1) = t_2(cn_2) \), \( o_2(cn_1) = o_2(cn_2) \), and \( m_1(cn_1) = m_1(cn_2) \).

2. Let \( cn_1 \equiv_2 cn_2 \) if and only if there are \( cn'_1 \), \( cn'_2 \) such that \( A_2(cn_1) \rightsquigarrow cn'_1 \mathcal{R}_2 cn'_2 \leftarrow A_2(cn_2) \).

Clearly, \( \equiv_2 \) identifies more extended configurations than \( \equiv_1 \).

Corollary 9.13. \( \equiv_1 \subseteq \equiv_2 \)

Proof. If \( t_1(cn_1) = t_1(cn_2) \) then \( t_2(cn_1) = t_2(cn_2) \) and similar for \( o_1 \) and \( o_2 \). The result follows.

We also obtain that normalization respects normal form equivalence.

Corollary 9.14. If \( A_2(cn) \rightsquigarrow cn' \) then \( cn \equiv_2 cn' \)

Proof. By prop. 9.11.

The proof of reduction closure follows that of lemma 9.6 quite closely.

Lemma 9.15. \( \equiv_2 \) is reduction closed.

Proof. See appendix 3.

Proposition 9.16. \( \equiv_2 \) is a type 2 witness relation

Proof. Similar to the proof of prop. 9.7.

It follows that if \( cn_1 \equiv_2 cn_2 \) then \( cn_1 \cong cn_2 \).

Corollary 9.17. If \( A_2(cn) \rightsquigarrow cn' \) then \( cn \equiv cn' \)

Proof. None of the rules used in alg. 2 affects the shape of the normal form. Thus, if \( A_2(cn) \rightsquigarrow cn' \) then \( cn \equiv_2 cn' \). But then \( cn \equiv cn' \), by prop. 9.16.

10 Correctness

In this section we prove correctness of the network semantics by mapping each well-formed type 1 configuration \( \text{bind } z.cn \) in standard form to a well-formed type 2 configuration \( \text{down}(cn) \) on an arbitrary underlying network graph. We then prove that the two configurations are barbed equivalent, i.e. that \( \text{bind } z.cn \cong \text{down}(cn) \).

Defining the Underlying Network Graph We first fix a graph represented as a well-formed type 2 configuration \( cn_{\text{graph}} \) with a distinguished UID \( u_0 \), similar to the way initial configurations are defined in section 6. Thus, \( cn_{\text{graph}} \) consists of nodes and links only, each node \( u \) in \( cn_{\text{graph}} \) has the form \((u, t)\), and each link has the form \((u, \varepsilon, u')\). The routing tables \( t \) are defined later.
Representing Names and Values  To represent names, one complication is that names in the type 1 semantics need to be related to names in the type 2 semantics, which does not include the binding construct of the type 1 semantics, but on the other hand has different generator functions (the functions newf and newo). This prevents the name relation from being modeled using the identity relation. To address this we assume that names and futures in the type 1 semantics are really symbolic, connected to concrete names/futures used in the type 2 semantics by means of an injective \(\text{rep}\) representation map, taking internal names \(f\), \(o\) in the type 1 semantics to names \(\text{rep}(f)\), \(\text{rep}(o)\) in the type 2 semantics. For convenience we extend the name representation map \(\text{rep}\) to arbitrary values and task environments as follows:

- \(\text{rep}(o) = o\), if \(o \in \text{Ext}\),
- \(\text{rep}(p) = p\), if \(p \in \text{PVal}\),
- \(\text{rep}(l)(x) = \text{rep}(l(x))\), \(\text{rep}(l)(\text{ret}) = \text{rep}(l(\text{ret}))\)

Representing Object Environments  To extend \(\text{rep}\) also to object environments a complication is that object environments in the type 2 semantics must be defined partially in terms of the type 1 environments (for object variables) and partially in terms of the future containers available in the “root configuration”, since the type 1 semantics uses future containers in place of forwarding lists stored in object environments. To this end we first define an auxiliary function \(\text{oenvmap}(cn, p, \text{rep}) \in \text{Fut} \rightarrow \text{Val}_\bot\) on triples of type 1 configurations, a pool of OID/future constants, and a name representation map, as a function which gathers together assignments to futures as determined by the future containers in \(cn\):

- \(\text{oenvmap}(0, p, \text{rep})(f) = \text{oenvmap}(\text{tsk}, p, \text{rep}) = \text{oenvmap}(\text{obj}, p, \text{rep}) = \bot\)
- \(\text{oenvmap}(f(f, v_\bot), p, \text{rep})(f') = \text{if } \text{rep}(f) = f' \text{ then } \text{rep}(v_\bot) \text{ else } \bot\)
- \(\text{oenvmap}(\text{bind } o.\text{cn}, p \cup \{o'\}, \text{rep})(f) = \text{oenvmap}(cn, p, \text{rep}[o' / o])(f)\)
- \(\text{oenvmap}(\text{bind } f.cn, p \cup \{f'\}, \text{rep})(f'') = \text{oenvmap}(cn, p, \text{rep}[f'/f])(f'')\)
- \(\text{oenvmap}(cn_1 cn_2, p, \text{rep})(f) = \text{oenvmap}(cn_1, p, \text{rep})(f) \sqcup \text{oenvmap}(cn_2, p, \text{rep})(f)\)

Fix now a root type 1 configuration \(cn_0\) and a large enough pool \(p_0\) of names (proportional to the size of \(cn_0\), and computed using newf\((u_0)\) and newo\((u_0)\) to conform with our naming policy). Assume that \(cn_0\) is in standard form, i.e. \(cn_0 = \text{bind } z_0.cn'_0\) where \(cn'_0\) does not have binders. Fix \(g = \text{oenvmap}(cn_0, p_0, \bot)\) and \(cn_{\text{graph}}\) as above. We can now extend \(\text{rep}\) to object environments by:

- \(\pi_1(\text{rep}(a))(x) = \text{rep}(\pi_1(a))(x)\)
\[
\pi_2(\text{rep}(a))(f) = \begin{cases} 
(g(f), \varepsilon) & \text{if } g(f) \neq \bot \\
(\bot, \text{OID}(cn_0)) & \text{otherwise}
\end{cases}
\]

Since we have left the nature of expressions unspecified we need to additionally assume that the representation map commutes with expression semantics in the following way, i.e. that for all \(e, a, l\) the following equation holds:

\[
\text{rep}(\hat{e}(a, l)) = \hat{e}(\text{rep}(a), \text{rep}(l))
\] (6)

**Proposition 10.1.** Fix a type 1 well-formed root configuration \(cn_0\) in standard form and pool \(p_0\) as above. Then \(\text{rep}(a)(f) = (v, \varepsilon)\) if and only if \(f(f, v) \preceq cn\).

**Proof.** By well-formedness the future, if it exists, is unique. Pick a name representation map \(\text{rep}\). Then \(\text{oenvmap}(cn_0, p_0, \text{rep})(f)\) is defined and equal to \(v\) if and only if \(f(f, v) \preceq cn_0\). This is easily seen by induction on the structure of \(cn_0\) \(\Box\)

**Representing Call Containers** Another complication is that we need an operation to represent a type 1 call container as a message in the type 2 semantics. This is done in the obvious way by the operation \(\text{send}\) as follows:

\[
\text{send}(c(o, o', f, m, v), o(a, a', q_{in}, q_{out}) \ cn)
= o(a, a', u, q_{in}, \text{enq}(\text{call}(o, o', f, m, v), q_{out})) \ cn
\] (7)

**Representing Configurations** Given a name representation map \(\text{rep}\) we can now define the representation of a type 1 configuration as a transformer on type 2 configurations, initially the underlying network graph, as the mapping \(\text{down}\) as follows:

- \(\text{down}(0, \text{rep})(cn) = cn\)
- \(\text{down}(cn_1 \ cn_2, \text{rep}) = \text{down}(cn_1, \text{rep}) \circ \text{down}(cn_2, \text{rep})\)
- \(\text{down}(t(o, l, s), \text{rep})(cn) = t(\text{rep}(o), \text{rep}(l), s) \ cn\)
- \(\text{down}(o(a, a), \text{rep})(cn) = o(\text{rep}(o), \text{rep}(a), u_0, \varepsilon, \varepsilon) \ cn\)
- \(\text{down}(f(f, v_{\bot}), \text{rep})(cn) = cn\)
- \(\text{down}(c(o, o', f, m, v), \text{rep})(cn) = \text{send}(c(o, o', f, m, v), cn, u_0)\)

In other words, we represent type 1 configurations by first assuming some underlying network graph, secondly distributing the (centralized) assignments to futures in each object environment with the trivial forwarding lists, and then, once this is done, mapping the containers individually to type 2 level.
Defining Routing Tables The only detail remaining to be fixed above is the routing tables. For $u_0$ the initial routing table, $t_0$, needs to register all objects in $cn_0$, i.e.

$$t_0 = \text{reg}(g(o_0), u_0, \text{reg}(g(o_1), u_0, \text{reg}(..., \text{reg}(g(o_n), u_0, \perp)) \cdot \cdot \cdot )$$

where $o_0, \ldots, o_n$ are the OID's in $cn'_0$. For nodes $n(u,t)$ where $u \neq u_0$ we let $t$ be determined by some stable routing. This is easily computed using alg. 1, and we leave out the details.

**Definition 10.2 (Representation Map $\text{down}$).** Let a network graph $cn_{\text{graph}}$ and a name representation map $\text{rep}$ be given. For each well-formed type 1 configuration $cn_0$, the type 2 representation of $cn_0$ is the configuration $\text{down}(cn_0) = \text{down}(cn_0, \text{rep})(cn_{\text{graph}})$.

In this definition, forwarding lists are overapproximated as compared to the type 2 semantics, which forward futures only to objects that have actually received them. To handle this slight complication we need a little lemma saying that for well-formed type 2 configurations, forwarding lists can be extended without affecting observable behaviour. To make this precise say that $o(a, u, q_{\text{in}}, q_{\text{out}}) \text{ extends } o'(a', u', q'_{\text{in}}, q'_{\text{out}})$, if $o = o'$, $u = u'$, $q_{\text{in}} = q'_{\text{in}}$, $q_{\text{out}} = q'_{\text{out}}$, $a(x) = a'(x)$ for all $x$, and $\pi_1(a(f)) = \pi_1(a'(f))$ and $\pi_2(a(f)) \supseteq \pi_2(a'(f))$ for all $f$.

**Lemma 10.3.** Suppose that $cn$ is WF2, and $cn'$ differs from $cn$ only by replacing each object $obj$ by an object $obj'$ such that $obj'$ extends $obj$. Then $cn'$ is WF2 as well, and $cn \equiv cn'$.

**Proof.** The check that $cn$ is WF2 only if $cn'$ is, is straightforward. Relate $cn$ and $cn'$ by the relation $R$ such that each object $obj$ in $cn$ is replaced by objects $obj'$ in $cn'$ such that $obj'$ extends $obj$, or vice versa. We show that $R$ is a type 2 witness relation. Evidently $R$ is symmetric. For the remaining properties (reduction closure and barb preservation) it suffices to note that $\{cn_1 \mid \mathcal{A}_2(cn) \rightsquigarrow cn_1\} = \{cn_2 \mid \mathcal{A}_2(cn') \rightsquigarrow cn_2\}$. This property relies heavily on Future Liveness, def. 7.4. This establishes the result. \hfill \Box

Following [11] we can now show a key lemma allowing us to relate transitions in the two semantics under barbed equivalence.

**Lemma 10.4.** Let bind $z.cn$ be WF1 in standard form.

1. If bind $z.cn \rightarrow bind z'.cn'$ then $\text{down}(cn) \rightarrow^* o \equiv \text{down}(cn')$

2. If $\text{down}(cn) \rightarrow cn''$ then for some $z'$, $cn'$, bind $z.cn \rightarrow^* bind z'.cn'$ and $cn'' \equiv \text{down}(cn')$

**Proof.** See appendix 4. \hfill \Box

We can now state the main result.

**Theorem 10.5** (Correctness of the Type 2 Semantics). For all well-formed type 1 configurations $cn$ on a connected network graph,

$$cn \equiv \text{down}(cn)$$
Proof. We exhibit a barbed bisimulation relation \( R \) as follows:

\[
R = \{(cn, cn') \mid \downarrow(cn) \sim= cn'\}
\]  

We show that \( R \) is a witness relation.

First for reduction-closure: If \( cn_1 R cn_2 \) then \( \downarrow(cn_1) \sim= cn_2 \). If \( cn_1 \rightarrow cn_1' \) then by lemma 10.4.1, \( \downarrow(cn_1) \rightarrow^* cn' \sim= \downarrow(cn_1') \). It follows that \( cn_2 \rightarrow^* cn_2' \) such that \( cn' \sim= cn_2' \). But then \( cn_1' R cn_2' \). Conversely, if \( cn_2 \rightarrow cn_2' \) then \( \downarrow(cn_1) \rightarrow^* cn' \) and \( cn' \sim= cn_2' \). By lemma 10.4.2, \( cn_1 \rightarrow^* cn_1' \) and \( cn' \sim= \downarrow(cn_1') \). But then \( cn_1' R cn_2' \) as desired.

For Barb Preservation assume \( cn_1 R cn_2 \). Then \( cn_1 \downarrow \text{obs} \) if and only if \( \downarrow(cn_1) \downarrow \text{obs} \) if and only if \( cn_2 \downarrow \text{obs} \).

\[ \Box \]

11 Scheduling

The type 2 semantics is highly nondeterministic. The semantics says nothing about how frequently routing tables are to be exchanged, when messages should be passed between the different queues, when future messages are to be sent, and when, and to where, objects are to be transmitted. Resolving these choices is a crucial tradeoff between management overhead and performance. For instance, if routing tables are exchanged at a very high frequency, routing can be always assumed to be in stable state. This ensures short end to end routes, but at the expense of a large management overhead. This raises the question of how to determine these parameters, something which we address in more detail in [12].

Regardless how this is done, a real implementation needs to resolve these choices. This is tantamount to eliminating nondeterminism from the type 2 semantics, essentially by removing transitions. Thus, in a sense, theorem 10.5 achieves more than is called for, as soundness and full abstraction a priori applies only to the type 2 semantics with all transitions included.

A scheduler can be viewed abstractly as a predicate on histories in the following way. Let a scheduled execution be any sequence \( \rho = cn_0 \cdot cn_1 \cdots cn_n \) such that \( cn_i \rightarrow cn_{i+1} \) for all \( i : 0 \leq i < n \) where the \( cn_i \) are well-formed type 2 configurations. A scheduler is a predicate \( S \) on such sequences, with the property that

1. \( S(\langle cn \rangle) \) for any \( cn \) where \( \langle cn \rangle \) is the one element execution consisting of \( cn \) (a scheduler kicks in only when execution is started), and

2. if \( S(cn_0 \cdots cn_n) \) and \( cn_n \rightarrow cn_{n+1} \) for some \( cn_{n+1} \) then \( S(cn_0 \cdots cn_n cn_{n+1}) \)

for exactly one \( cn_{n+1} \).

Then a scheduled type 2 semantics is a transition system on executions \( \rho = cn_0 \cdots cn_n \) such that \( \rho \rightarrow \rho' \) if and only if \( \rho' = cn_0 \cdots cn_n cn_{n+1} \) and \( S(\rho') \).

Define now the barbed simulation preorder \( \sqsubseteq \) on executions by requiring the existence of a witness relation \( R \) which satisfies reduction closure and barb preservation (when \( cn_0 \cdots cn_n \downarrow \text{obs} \) if and only if \( cn_n \downarrow \text{obs} \), but not necessarily symmetry. We immediately obtain from theorem 10.5 the following corollary:
Corollary 11.1. For all well-formed type 1 configurations $cn$ on a connected network graph,

$$\langle \text{down}(cn) \rangle \sqsubseteq cn .$$

Proof. It suffices to note that if $\rho = cn_0 \cdots cn_n \rightarrow \rho' = cn_0 \cdots cn_{n+1}$ then $cn_n \rightarrow cn_{n+1}$, and if $\rho \downarrow \text{obs}$ then $cn_n \downarrow \text{obs}$ as well. $\square$

12 Concluding Remarks

The contribution of the present paper has been to show that, using location independent routing, it is possible to devise novel and elegant network-based execution models for object-oriented languages with fairly sophisticated features such as futures, and with attractive properties regarding correctness, performance, and scalability. The present paper focuses on correctness. In other ongoing work [12] we study the use of the model presented here adapted to the full asynchronous fragment of the core ABS language [23] studied in the EU FP7 HATS project. In that paper we report on experimental results on autonomous performance adaptation for load balancing and latency management, with very promising results.

The correctness analysis is based on barbed equivalence, similar to other past work mostly belonging to the $\pi$-calculus school of process algebra [16, 9, 17]. A closely related precursor is Nomadic Pict [34]. In comparison with that work we obtain a simpler and in our opinion more elegant correctness treatment, chiefly because our solution obviates the need for locking and consequently preemption, which has well-known detrimental consequences in a bisimulation-oriented setting. Other related works include JoCaml [10] which also uses forward chaining, along with an elaborate mechanism to collapse the forwarding chains. In the Klaim project [1] compilers were implemented and proved correct for several variants of the Klaim language, using the Linda tuple space communication model and a centralized name server to identify local tuple servers. The Oz kernel language [35] uses a monotone shared constraint store in the style of concurrent constraint programming. The Oz/K language [26] adds to this a notion of locality with separate failure and mobility semantics, but no real distribution or communication semantics is given (long distance communication is reduced to explicit manipulation of located agents, in the style of Ambient calculus [4]). Past correctness analyses for languages with futures include [8] which prove a confluence result for their language of asynchronous sequential processes, however without an explicit treatment of distribution, communication, and routing.

Substantial work has been going on in the HATS project on the ABS language and its extensions, for instance towards software product lines [33]. In a sequence of papers, Johnsen and coworkers, cf. [24], have studied an extension of ABS with deployment components that can be used as a reflection mechanism for explicit performance management. The use of reflection allows many aspects of resource control to be brought into the ABS framework and managed as part of the regular application develop-
ment process. By contrast, our focus is on algorithmics and the automation of the performance adaptation processes, and for this type of application reflection is not a main concern.

Scalability is not fully resolved in the present work. We use a rather naive distance vector routing scheme based on Bellman-Ford. Distance vector routing has unit stretch but is not compact: Routing tables need to contain on the order of one entry per network node. This is no problem for networks of moderate size, but for scalability other routing schemes are needed, as outlined in the introduction. The Bellman-Ford scheme has other well-known drawbacks that arise in the case of intermittent network partitioning.

Besides routing we see two main directions for future work. The first is to examine richer language semantics, specifically towards more dynamicity. In ongoing work we are studying power control: Adding an explicit knob to the network semantics for turning nodes on and off. Further down the line it is of interest to consider both crash failures and byzantine failures. The second, parallel, avenue is to study performance adaptation in richer and more realistic settings. In [12] our only management knob is object migration, and the management objective is to obtain good load balancing combined with good clustering properties. However, a real implementation, in particular when combined with network dynamicity will have many more management knobs such as buffer sizes, processor load, and power control. How to effectively control object network performance in such a multi-dimensional setting is a significant future challenge.

References


Appendix 1: Proofs for Section 5

Proposition 5.2 The identity relation is a type 1 witness relation. Barbed equivalence is a type 1 witness relation. If $R, R_1, R_2$ are type 1 witness relations then so is

1. $R^{-1}$
2. $R^*$
3. $R_1 \circ R_2 \circ R_1$

Proof. The identity relation is trivially symmetric, reduction closed and barb preserving. Suppose $cn_1 \cong cn_2$. Then $cn_1 \mathcal{R} cn_2$ for some type 1 witness relation. Then $cn_2 \mathcal{R} cn_1$ by symmetry, such that $cn_2 \cong cn_1$, so symmetry holds. If $cn_1 \rightarrow cn'_1$ then we find $cn'_2$ such that $cn_2 \rightarrow^* cn'_2$ and $cn'_1 \mathcal{R} cn'_2$. But then $cn'_1 \cong cn'_2$, and we have shown reduction closure. For barb preservation, if $cn_1 \downarrow obs$ then $cn_2 \downarrow obs$, as $\mathcal{R}$ is barb preserving. Inverse closure follows from symmetry, and reflexive, transitive closure follows by a straightforward inductive argument. For property 3, if $cn_1 \mathcal{R}_1 \circ \mathcal{R}_2 \circ \mathcal{R}_1 cn_2$ then $cn_1 \mathcal{R}_1 cn_{1,1} \mathcal{R}_2 cn_{1,2} \mathcal{R}_1 cn_2$ for some $cn_{1,1}, cn_{1,2}$ so $cn_2 \mathcal{R}_1 cn_{1,1} \mathcal{R}_2 cn_{1,2} \mathcal{R}_1 cn_1$ by symmetry of $\mathcal{R}_1$ and $\mathcal{R}_2$. Second, if $cn_1 \rightarrow cn'_1$ we find $cn'_{1,1}, cn'_{1,2}, cn'_{1,2}$ such that $cn_{1,1} \rightarrow^* cn'_{1,1}$ etc. such that, by stepwise iteration, $cn'_1 \mathcal{R}_1 \circ \mathcal{R}_2 \circ \mathcal{R}_1 cn'_2$, showing reduction closure. Barb preservation is similar, but simpler. □

Proposition 5.4 Suppose $cn$ is WF1, $o \notin fn(cn)$, and $cn \circ(o,a)$ is WF1. Then $cn \cong cn \circ(o,a)$

Proof. Let $cn \mathcal{R} cn'$ if and only if $cn' = cn \circ(o,a)$ as described in the the statement of the proposition. We show that $\mathcal{R} \cup \mathcal{R}^{-1}$ is a type 1 witness relation:

Symmetry: Trivial

Reduction closure: First if $cn \rightarrow cn_1$ then $cn \circ(o,a) \rightarrow cn_1 \circ(o,a)$, and $cn_1 \mathcal{R} cn_1 \circ(o,a)$. If $cn \circ(o,a) \rightarrow cn_1$ and $cn_1$ does not have the required shape $cn'_1 \circ(o,a)$ such that $cn \rightarrow cn'_1$ then the transition must be an instance of call, and $o$ must be the callee. This is so, as in any other case, a transition involving $o$ requires a task at $o$ to be present in $cn$. But then $o$ is free in $cn$ as well, contrary to assumptions.

Barb preservation: If $cn \downarrow obs$ then $cn \circ(o,a) \downarrow obs$, and it $cn \circ(o,a) \downarrow obs$ then $cn \downarrow obs$, as $o \notin fn(cn)$. □

Appendix 2: Proofs for Section 7

Lemma 7.3 Fix $cn$ and an object $\text{obj} = o(a,u,q_{in},q_{out}) \leq cn$. If $o$ is on the notification path of $f$ in $n$ steps in the configuration $cn$, $cn \rightarrow cn'$, then $o$ is on the notification path of $f$ in at most $n$ steps in $cn'$. 

39
Proof. The proof is by induction on \( n \). We follow the case analysis in def. 7.2. In case 1 we obtain that \( \pi_1(a(f)) \neq \bot \) and then by prop. 6.2.3 we find \( obj' \) such that \( \pi_1(a'(f)) \perp \). In case 2, by prop. 6.2.4 there are two options: Either we find a task \( t(o, l', s') \preceq cn' \) with \( l'(ret) = f \) or else we find \( obj' \) such that \( \pi_1(a'(l'(ret))) \perp \). In case 3 we obtain that either future \( (o, f, v) \preceq cn' \), or else \( \pi_1(a'(f)) = v \). In case 4 we find an object \( obj'' = o(\omega''', a'', u'', q''_{in}, q''_{out}) \preceq cn \) such that \( \omega'' \) is on the notification path of \( f \) in \( n - 1 \) steps in \( cn' \). By prop. 6.2.3 we find the derivative \( obj''' = o(\omega''', a''', u''', q'''_{in}, q'''_{out}) \preceq cn' \) of \( obj'' \), and by the induction hypothesis \( \omega'' \) is also on the notification path of \( f \) in \( cn' \), now in some \( n'' \leq n - 1 \) steps. By inspection of the rules we see that either \( \pi_2(a''''(f)) \) is a suffix of \( \pi_2(a''''(f)) \), or else there is a message future \( (o, f, \pi_3(a''''(f))) \preceq cn' \). In either case we can conclude. Finally, in case 5, by inspection of the rules we either find that the call message \( call(o, o''', f, m, v) \preceq cn' \), or else there is a task \( t(o''', u''', v) \preceq cn' \) such that \( l''(ret) = f \), and we are done. \( \square \)

Lemma 7.5 If \( cn \) is WF2 and \( cn \rightarrow cn' \) then \( cn' \) is WF2.

Proof. We consider each transition rule in turn. For well-formedness, NID and link uniqueness is trivial, as \( graph(cn) = graph(cn') \).

OID uniqueness: In all rules except new-2 there is a 1-1 correspondence between object occurrences in \( cn \) and object occurrences in \( cn' \). This is sufficient to conclude. For new-2 it is sufficient to note that \( o' \) is a freshly generated OID.

Object-Node Existence, Task-Object Existence: The properties follow since nodes and objects are never removed. In the first instance objects can only be created when the node is present, and in the second instance tasks can only be created when the object is present.

Object Existence, Object Nonexistence: Simple invariant checks.

Buffer cleanliness: We check that only object bound messages enter in- and out-queues. This concerns the rules msg-rcv, call-send, fut-send, msg-delay-1 and msg-delay-2 only. The check is routine.

Local Routing Consistency, 1, 2: Easily proved by case analysis.

Future uniqueness: We only need to consider rules which assign a non-\( \bot \) value to futures. This happens in rules fut-send, fut-rcv and ret-2. The former two rules are immediate, and for ret-2 we use the assumption that \( cn \) satisfies 7.4.5.

Single writer: Again we only need to consider the rules fut-send, fut-rcv and ret-2. Since for the former two rules \( cn \) assigns \( v \) to \( f \) if and only if \( cn' \) does so, only ret-2 remains, which is immediate.

Future liveness: Let \( o(o, a', u', q'_{in}, q'_{out}) \) be the derivative of \( o(o, a, u, q_{in}, q_{out}) \) in \( cn \), and assume that \( f \) is pending in \( o \) for the configuration \( cn' \). If \( o \) is a newly created OID, no future is pending in \( o \). Either \( a'(f) \perp \), or else a call message \( call(o', a', f', m, v) \) is in transit in \( cn' \) and \( f \) occurs in \( v \). We proceed by cases on the transition rule leading to \( cn' \). Any rule that does not directly affect any of the conditions in def. 7.1 or def. 7.2 immediately allows to conclude that \( f \) is pending in \( o \) also in \( cn \). By the induction hypothesis we
can conclude that \(o\) is on the notification path from \(f\) in \(cn\) and then \(o\) is on the notification path from \(f\) also in \(cn'\), as the only exception in lemma 7.3 is when \(o\)'s environment is updated. For the remaining rules there are the following cases to consider:

- **call-send**: Assume first that \(o\) is the sending object. Either \(f\) is the newly introduced future in which case \(o\) is on the notification path from \(f\) according to 7.2.5, since a call message is in transit from \(o\) to \(o'\) with return future \(f\). If \(f\) is another future which is pending in \(o\) in \(cn'\) then \(f\) is also pending in \(o\) in \(cn\). By the induction hypothesis, \(o\) is on the notification path from \(f\) in \(cn\). Then \(o\) is also on the notification path from \(f\) in \(cn'\) by lemma 7.3. On the other hand if \(o\) is not the sending object the case is immediately closed by the induction hypothesis, as the “pending” relation transfers from \(cn'\) to \(cn\). We then apply the IH to conclude that \(o\) is on the notification path from \(f\) in \(cn\), and then we use lemma 7.3 to conclude that this also applies to \(cn'\).

- **call-rcv**: Assume first that \(o\) is the object receiving the call, and that \(f\) is the future of the call. Then \(o\) is on the notification path from \(f\) by def. 7.2.2. Another option is that \(f\) is a future in \(v\) (referring to the transition rule in fig. 9). Then \(f\) is pending in \(o\) in \(cn\) as well, by def. 7.1. If \(f\) is some other future the case is completed by the IH as above.

- **fut-send**: Follows from the induction hypothesis and lemma 7.3, as in the case for call-send.

- **fut-rcv**: If \(f\) is pending in \(o\) for the configuration \(cn'\) then \(f\) is pending in \(o\) also for \(cn\), and \(f\) is not the received future. But then the result follows by the IH and lemma 7.3.

- **ret-2**: If \(f\) is pending in \(o\) for \(cn'\) then \(f\) is not the return future and we again complete by the induction hypothesis.

\(\Box\)

### Appendix 3: Proofs for Section 9

**Proposition 9.2**  Algorithm 1 terminates.

*Proof.* In each iteration of the outermost loop of alg. 1, exactly one message is enqueued on each proper link, and at least one message is dequeued (from all link queues), so the sum of messages in transit in link queues does not exceed its initial value. The rules msg-rcv, msg-delay-1, obj-rcv cause messages to leave the link queues, except for external messages. Moreover, if the link queues have only routing table messages the algorithm terminates in that iteration. So if the algorithm fails to terminate it must be because msg-route is from some point \(n_0\) onwards applied in each iteration of the outermost loop. From \(n_0\) onwards, no messages other than table updates are delivered (to the receiving node, or to the receiving object). In particular, no object messages can be in transit on a link from that point onwards.
We show that routing tables must at some point stabilize. At point $n_0$ (as all other points) each node $u$ has $t(o) = (u, 0)$ whenever $o$’s host is $u$, by def. 7.4.9. Let $m_0$ be the length of the largest link queue at the point from which no messages are delivered. After $n_0 + m_0 + 1$ iterations, each node $u$ has received at least one table update from each of its neighbours $u'$, and the last table update applied to $u$ has $t(o) = 0$. As result, at point $n_0 + m_0 + 1$ each node $u$ has $t(o) = (u', 1)$ whenever the host of $o$ is $u'$ and the minimal length path from $u$ to $u'$ has length 1. The entry of $u$’s routing table for $o$ will not change from that point onwards. We say that those entries are stable. Proceeding, let $m_1$ be the length of the largest link queue at point $n_0 + m_0 + 1$. After $n_0 + m_0 + 1 + m_1 + 1$ iterations each routing table entry with length 2 (or less) will be stable. In the limit each entry will be stable. It follows that algorithm 1 must terminate, since, once routing has stabilized, rule msg-route can only be applied a finite number of times before the message will be delivered.

The only detail remaining to be checked is that a message can always be read from a link, but table and object messages can always be delivered, and call and future messages can also always be delivered, if nothing else to the self loop, in case the routing table has not yet been updated, or if the message is external and the destination object is not known to the routing table. This is the only case where msg-delay-1 is used, in fact.

\[\text{Lemma 9.6} \quad \equiv_1 \text { is reduction closed.}\]

\[\text{Proof. Assume that} \quad cn_1 \rightarrow cn'_1 \quad (9)\]

\[\text{and} \quad cn_1 \equiv_1 cn_2, \quad (10)\]

\[\text{and show that} \quad cn_2 \rightarrow^* cn'_2 \quad (11)\]

\[\text{such that} \quad cn'_1 \equiv_1 cn'_2. \quad (12)\]

The proof is by cases on the rewriting rule applied. Most cases are straightforward. For rules not among call-send-in, call-send-out, call-rcv, fut-send, fut-rcv, if

\[A_1(cn_1) \rightsquigarrow cn_{1,1} \quad R_1 \quad cn_{2,1} \rightsquigarrow A_1(cn_2) \quad (13)\]

and $A_1(cn'_1) \rightsquigarrow cn'_{1,1}$ then we obtain

\[cn'_{1,1} \quad R_1 \quad cn_{1,1}, \quad (14)\]

by prop. 9.3 and since the correspondences between tasks, objects, and call and future messages are maintained in pre- and poststates. For the remaining rules:
call-send: From (9) we may assume that
\[ cn_1 = cn_{1,1} o (o,a,u,q \text{ in}, q \text{ out}) t (o,l,x = e_1!m(e_2); s) \] (15)
\[ cn'_1 = cn_{1,1} o (f w(v,v', \text{ init}(f,a)), u, q \text{ in}, \text{ cnq}(\text{ msg}, q \text{ out})) t (o,l,f[x], s) \] (16)
where \( v = e_2(a,l), f = \text{ newf}(u), \text{ msg} = \text{ call}(o,d', f, m, v), d' = e_1(a,l) \). From (10) we get
\[ A_1(cn_1) \sim cn''_1 R_1 cn''_2 \sim A_1(cn_2) . \] (17)
By prop. 9.3, (15) and (17),
\[ o(o,a,u,q \text{ in}, q \text{ in}, q \text{ out}), t(o,l,x = e_1!m(e_2); s) \in cn''_1 \] (18)
and hence, by the definition of \( R_1 \),
\[ o(o,a,u,q \text{ in}, q \text{ in}, q \text{ out}), t(o,l,x = e_1!m(e_2); s) \in cn''_2 \] (19)
as well. But then it follows that the configuration \( cn_2 \) can mimic the call-send step by \( cn_1 \) by first stabilizing to \( cn''_2 \) and then performing the call-send step, obtaining \( cn'_2 \). It is also clear by prop. 9.3 that (12) holds, completing the case.

The remaining cases are involved a similar amount of detail but are, as this, essentially straightforward.

\[ \square \]

**Proposition 9.9** Algorithm 2 terminates.

**Proof.** Routing is stable after each run of alg. 1, and none of the rules applied in the outermost loop in the first outermost loop affect routing. Thus, one of \( \text{ msg-send } \) or \( \text{ msg-delay-2 } \) will be enabled whenever the output queue is nonempty, causing output queue size to decrease by one. By Buffer Cleanliness, one of \( \text{ call-rcv } \) or \( \text{ fut-rcv } \) will be applicable if the object in-queue is nonempty, decreasing in-queue size by one. Thus, when the inner while loop is reached, each nonempty inqueue has decreased in size by one, and each outqueue may have increased in size by one if the in-queue head position contains a delayed message.

Sending future messages may cause out-queues to increase in size. Each application of \( \text{ fut-send } \) causes a forwarding list to decrease in length by one. Thus termination of the inner while loop is clear. We need to argue that the outer while loop also terminates.

We first show that, eventually, no forwarding list is incremented. Only two rules can cause forwarding lists to increase in size, namely call-send and call-rcv. Of these, call-send is never used in either alg. 1 or alg. 2. Each application of call-rcv consumes one call message, and none of the rules cause new call messages to be created. Thus, eventually, call-rcv is never applied, and from that point onward forwarding lists are either emptied completely by the inner loop, or they remain untouched, since their corresponding future is undefined. Futures can become instantiated by fut-rcv, but again, this can only happen a bounded number of times. Moreover,
the only rule causing futures to be created is msg-send, so the supply of
futures to consider is fixed. Consequently, eventually, each future either
remains uninstantiated forever, or else the corresponding forwarding list is
empty. From that point onward, no fut-send is enabled, and the innermost
loop terminates trivially in all future iterations. In this situation, since msg-
send, call-rcv, and fut-rcv all consume messages from a bounded resource
(the set of messages in transit), if the outermost loop fails to terminate the
only option is that, from some point onwards, only msg-delay-2 is applied.
From this point onward, since routing is stable, all messages will eventually
be delivered.

Termination of the final loop is trivial. Observe that alg. 2 does not rely
on routing to move the object towards u. For the algorithm it is sufficient to
establish that some good direction exists, and this is clearly the case as the
network is stable.

\begin{proposition}
Suppose \(cn\) is WF2. If \(A_2(cn) \Rightarrow cn'\) then
\begin{enumerate}
\item \(cn'\) is in normal form
\item \(\text{graph}(cn) = \text{graph}(cn')\)
\item \(t_2(cn) = t(cn')\)
\item \(o_2(cn) = o(cn')\)
\item \(m_1(cn) = m(cn')\)
\end{enumerate}
\end{proposition}

\begin{proof}
Property 9.11.2 follows from prop. 9.3.1.

For property 9.11.3 observe first that the function \(t_2\) is invariant under tran-
sitions used in alg. 2. On termination of alg. 2 only external messages are
in transit, and since no rule causes a task to be modified, 9.11.3 follows.

For 9.11.4 let \(o(o_2,a_2,u_2,q_{in,2},q_{out,2}) \in o(cn')\). We show
\[
o(o_2,a_2,u_2,q_{in,2},q_{out,2}) \in o_2(cn).
\]
By definition, \(q_{in,2} = q_{out,2} = \varepsilon\). Also, \(u_2 = u\). We know that there is an
object container \(o(o_1,a_1,u_1,q_{in,1},q_{out,1}) \preceq cn\), as there is a 1-1 correspondence
between object containers in pre- and poststate for each transition used in
alg. 2. We also know that \(a'(x) = a_2(x)\) for all \(x\). Suppose finally that an
object container \(ct = o(a_1,a_1,u_1,q_{in,1},q_{out,1})\) exists in \(cn\) with \(a_1(f) = (v, o)\).
Let \(o(a_1,a'_1,u'_1,q'_{in,1},q'_{out,1})\) be the derivative of \(ct\) in \(cn'\). Then \(\pi_1(a'_1(f)) = v\)
as well, by prop. 7.7. We know by Future Uniqueness that \(a_2(f) = (v', o')\)
implies \(v' = v\). It remains to show that \(\pi_1(a_2(f)) \neq \bot\). Assume not. The
future \(f\) is then pending in \(a_2\). By the Type 2 Subject Reduction Lemma,
7.5, \(cn'\) is WF2, so by Future Liveness, \(a_2\) is on the notification path from \(f\)
in \(cn'\) in some \(n\) steps. We proceed by induction on \(n\):
\begin{itemize}
\item \(n = 0\) and \(\pi_1(a_2(f)) = v \neq \bot\). This is a contradiction.
\item \(n = 1\) and there is a task \(t(o_2,l_2,s_2) \preceq cn'\) with \(l_2(\text{ret}) = f\). By Future
Binding, \(\pi_1(a'_1(f)) = \bot\), a contradiction
\end{itemize}
\end{proof}
• \( n = 1 \) and there is a future message future\((o_2, f, v') \preceq cn'\), which is a contradiction, since the only queued messages in \( cn' \) are external

• \( n = n' + 1 \) and there is an object \( o(o, a, u, q_{in}, q_{out}) \preceq cn' \) such that \( o_2 \in \pi_2(a(f)) \), and \( o \) is on the notification path from \( f \) in \( n' \) steps. Either \( \pi_1(a(f)) = v \neq \bot \), but then \( cn' \) is not in normal form, contradiction. Alternatively, we conclude by the IH.

• There is a call message call\((o_2, o, f, m, v) \preceq cn'\). This again contradicts the external messages only property.

We can thus conclude that \( o(o_2, a_2, u_2, q_{in}, q_{out}) \in o_2(cn) \). Conversely, assume that \( o(o_2, a_2, u_2, q_{in}, q_{out}) \in o_2(cn) \). Object \( o_2 \) has exactly one derivative in \( cn' \), by well-formedness. That object has empty queues, the same UID as in \( cn \), preserves assignments to variables, and has \( a_2(f) \) assigned to a non-\( \bot \) value if and only if some object in \( cn' \) has so, by the above argument.

For 9.11.5 the property holds as it does so already for alg. 1.

We finally need to prove 9.11.1. Property 9.10.1 is trivial, as each run of alg. 2 ends with a run of alg. 1, and alg. 1 ensures that \( cn' \) has stable routing.

Property 9.10.2 holds since alg. 1 ensures almost empty link queues, and since on termination, alg. 2 ensures empty object queues. For 9.10.3 the result follows since only external messages are in transit in \( cn' \). For 9.10.4, if \( obj = o(o, a, u, \varepsilon, \varepsilon) \) satisfies the properties defining \( o_2 \) above then, referring to those conditions, \( u' = u'' = u, a' = a, q_{in} = \varepsilon = q_{out} \), and \( obj \in cn' \) as needed to be shown. Finally, 9.10.5 holds since it does so already for alg. 1.

**Lemma 9.15** \( \equiv_2 \) is reduction closed.

**Proof.** Assume that

\[
\text{cn}_1 \rightarrow \text{cn}'_1 \tag{20}
\]

and

\[
\text{cn}_1 \equiv_2 \text{cn}_2 , \tag{21}
\]

and we show that

\[
\text{cn}_2 \rightarrow^* \text{cn}'_2 \tag{22}
\]

for some \( \text{cn}'_2 \) such that

\[
\text{cn}'_1 \equiv_2 \text{cn}'_2 \tag{23}
\]

As above the proof is by cases on the rewrite rule justifying 20. We can assume that \( \text{cn}_2 \) is in normal form, by 9.14 and transitivity of \( \equiv_2 \). For rules that do not affect \( t_2(cn_1), o_2(cn_1), \) or \( m_1(cn_1) \) the result is trivial. Rules in fig. 5 commute directly, i.e. the same rule applied to \( cn_1 \) can be applied to \( cn_2 \), in the same way. This follows since \( cn_2 \) is in normal form. Rules such as msg-send, msg-rcv that ship around messages between object and link queues are also very easy to prove, by reference to prop. 21.
call-send: From (20) we may assume that

\[ cn_1 = cn_{1,1} o(o, a, u, q_{in}, q_{out}) t(o, l, x = e_1!m(e_2); s) \]  
(24)

\[ cn'_1 = cn_{1,1} o(o, fw(v, o', init(f, a)), u, q_{in}, enq(msg, q_{out})) t(o, l(f(x), s)) \]  
(25)

where \( v = e_2(a, l) \), \( f = newf(u) \), \( msg = call(o, o', f, m, v) \), \( o' = \hat{e}_1(a, l) \). By 21, since \( cn_2 \) is in normal form, we obtain

\[ A_2(cn_1) \Rightarrow cn''_1 R_2 cn_2 \]  
(26)

for some choice of \( cn''_1 \). By prop. 9.11 we obtain that

\[ cn_2 = cn_{2,1} o(o, a', u, \varepsilon, \varepsilon) t(o, l, x = e_1!m(e_2); s) \]  
(27)

where \( a'(x) = a(x) \) for variables \( x \), and \( a'(f) = (v', \varepsilon) \) if and only if \( a_1(f) = (v', o) \) for some \( o \), where \( a_1 \) is an object environment in \( cn_1 \). It follows that \( cn'_2 \) can be chosen so that

\[ cn'_2 = cn_{2,1} o(o, fw(v, o', init(f, a')), u, \varepsilon, enq(msg, \varepsilon)) t(o, l(f(x), s)) \]  
(28)

and

\[ A_2(cn'_1) \Rightarrow cn_{1,3} \]  
(29)

with \( cn_{1,3} = cn_{1,4} o(o, fw(v, o', init(f, a')), u, \varepsilon, enq(msg, \varepsilon)) t(o, l(f(x), s)) \) such that \( cn_{1,4} R_2 cn_{2,1} \). It follows that \( cn_2 \Rightarrow cn'_2 \) and \( cn'_1 \equiv cn'_2 \), as desired.

fut-send: From 20 we can assume

\[ cn_1 = cn_{1,1} o(o, a, u, q_{in}, q_{out}) \]  
(30)

\[ cn'_1 = cn_{1,1} o(o, a'[v/f], u, q_{in}, enq(\text{future}(o_1, f, v), q_{out})) \]  
(31)

where \( a(f) = (v, o_1 :: o_2) \). Using 21, since \( cn_2 \) is in normal form, we obtain

\[ A_2(cn_1) \Rightarrow cn''_1 R_2 cn_2 \]  
(32)

for some choice of \( cn''_1 \), and we can write \( cn''_1 \) in the form

\[ cn''_1 = cn''_{1,1} o(o, a', u, \varepsilon, \varepsilon) \]  
(33)

where \( a'(x) = a(x) \) and \( a'(f') = (v', \varepsilon) \) if and only if some object environment in \( cn_1 \) assigns \( v' \) to \( f' \). By prop. 9.11 we get

\[ cn_2 = cn_{2,1} o(o, a', u, \varepsilon, \varepsilon) \]  
(34)

and then we can choose \( cn'_2 \) as

\[ cn'_2 = cn_{2,1} o(o, a'[v/\varepsilon]/f], u, \varepsilon, enq(\text{future}(o_1, f, \varepsilon)) \]  
(35)

noting that then (23) holds.

call-rcv: In this case we get

\[ cn_1 = cn_{1,1} o(o, a, u, q_{in}, q_{out}) \]  
(36)

\[ cn'_1 = cn_{1,1} o(o, \text{future}(f, a'), init(v, init(f, a))), u, \text{deq}(q_{in}), q_{out}) t(o, \text{locals}(o, m, f, v), \text{body}(o, m)) \]  
(37)
where \( \text{hd}(q_{in}) = \text{call}(o', o, f, m, v) \). Again using prop. 21 with \( cn_2 \) in normal form we get that
\[
A_2(cn_1) \Rightarrow cn''_1 \mathcal{R}_2 cn_2
\]
(38)
where \( cn''_1 \) can be written as
\[
cn''_1 = cn''_1 o(o, a', u, \varepsilon, \varepsilon) t(o, locals(o, m, f, v), body(o, m))
\]
(39)
where \( a' \) is as in the previous case. Now using prop. 21, since \( \text{hd}(q_{in}) = \text{call}(o', o, f, m, v) \), we obtain
\[
cn_2 = cn_{2,1} o(o, a', u, \varepsilon, \varepsilon) t(o, locals(o, m, f, v), body(o, m)),
\]
(40)
and \( cn''_1 \mathcal{R}_2 cn_2 \), completing the case.

The remaining cases fut-rcv, ret-2, get-2, new-2, and the object migration rules are proved in a similar fashion as the above.

\[\square\]

**Appendix 4: Proofs for Section 10**

**Lemma 10.4** Let \( \text{bind } z. cn \) be WF1 in standard form.

1. If \( \text{bind } z. cn \rightarrow \text{bind } z'. cn' \) then \( \text{down}(cn) \rightarrow^* o \cong \text{down}(cn') \)

2. If \( \text{down}(cn) \rightarrow cn'' \) then for some \( z', cn' \), \( \text{bind } z. cn \rightarrow^* \text{bind } z'. cn' \) and \( cn'' \cong \text{down}(cn') \)

**Proof.** 1. We proceed by cases on the nature of the given type 1 transition. Let
\[
\text{bind } z. cn \rightarrow \text{bind } z'. cn'.
\]
(41)
Fix \( cn_{\text{graph}} \) and name representation map \( rep \). As above we elide uses of \( \text{ctxt-1} \) in both semantics by applying the rules to arbitrary configuration subsets, and we elide uses of \( \text{ctxt-2} \) in the type 1 semantics, by considering transitions in arbitrary binding contexts. Each of the remaining transitions in fig. 5 immediately translates into a corresponding transition at type 2 level, and moreover, the resulting type 2 configuration is in normal form.

Consider for instance rule \( \text{wfield} \). We obtain a type 1 transition of the form
\[
\text{bind } z. cn o(o, a) t(o, l, x = c; s) \rightarrow \text{bind } z. cn o(o, a[e(a, l)/x]) t(o, l, s)
\]
(42)
where $x \in \text{dom}(a)$. We obtain:

\[
\begin{align*}
\down(cn \ o(a, a) t(o, l, x = e; s)) &= (\down(cn, rep) \circ \down(o(a, a), rep) \circ \down(t(o, l, x = e; s), rep))(cn_{\text{graph}}) \\
&= \down(cn, rep)(\down(o(a, a), rep)(\down(t(o, l, x = e; s), \text{rep}))(cn_{\text{graph}})) \\
&= \down(cn, rep)(\down(t(o, l, x = e; s), \text{rep})(cn_{\text{graph}})) \\
&= \down(cn, rep)(\text{rep}(a), u_0, \varepsilon, \varepsilon) t(\text{rep}(a), \text{rep}(l), x = e; s) cn_{\text{graph}}) \\
&= \down(cn, rep)(\text{rep}(a), \text{rep}(a)[\hat{e}(\text{rep}(a), \text{rep}(l))/x], u_0, \varepsilon, \varepsilon) t(\text{rep}(a), \text{rep}(l), s) cn_{\text{graph}})
\end{align*}
\]

using (6) to justify the second but last step and the type 2 wfield rule to derive the transition.

The remaining rules in fig. 5 are proved in the same manner, so we proceed to the rules in fig. 6.

call: Consider the following type 1 transition:

\[
\begin{align*}
\text{bind } z. \text{cn } o(a, a) o(a', a') t(o, l, x = e_1 \text{ln}(e_2); s) \\
&\rightarrow \text{bind } z, f. \text{cn } o(a, a) o(a', a') t(o, l[f/x], s) \\
&\quad t(o', \text{locals}(o', m, f, e_2(a, l)), \text{body}(o', m)) \text{f}(f, \bot) \quad (43)
\end{align*}
\]

where $o' = \hat{c_1}(a, l)$. We calculate:

\[
\begin{align*}
\down(cn \ o(a, a) o(a', a') t(o, l, x = e_1 \text{ln}(e_2); s)) &= \down(cn, rep)(\down(o(a, a), rep)(\down(o(a', a'), rep)) \\
&\quad (\down(t(o, l, x = e_1 \text{ln}(e_2); s), \text{rep}))(cn_{\text{graph}})) \\
&= \down(cn, rep)(\text{rep}(a), u_0, \varepsilon, \varepsilon) o(\text{rep}(a'), \text{rep}(a'), u_0, \varepsilon, \varepsilon) t(\text{rep}(a), \text{rep}(l), x = e_1 \text{ln}(e_2); s) cn_{\text{graph}}) \\
&= \down(cn, rep)(\text{rep}(a), \text{rep}(a'), \text{rep}(f'), \text{rep}(v'), \text{init}(f', \text{rep}(a))), u_0, \varepsilon, \\
&\quad \text{enq}(\text{call}(\text{rep}(a'), \text{rep}(o'), f', m, \text{rep}(v'), \varepsilon)) \text{o}(\text{rep}(a'), \text{rep}(a'), u_0, \varepsilon, \varepsilon) t(\text{rep}(a'), \text{rep}(l)[f'/x], s) cn_{\text{graph}})
\end{align*}
\]

where by (6), $v = e_2(a, l)$, $o' = \hat{c_1}(a, l)$, $f' = \text{newf}(u_0)$, and where $\text{rep'} = \hat{c_1}(a, l)$.
rep[f′/f]. Using corollary 9.17 and the definition of ≡2 we obtain
\[cn′ \equiv \text{down}(cn, \text{rep}′(o)(\text{rep}'(a), a_1, u_0, \varepsilon, \varepsilon) \circ \text{rep}'(a'), a_2, u_0, \varepsilon, \varepsilon)\]
\[t(\text{rep}'(o'), \text{rep}'(l)[f'/x], s)\]
\[t(\text{rep}'(o'), \text{locals}(\text{rep}'(a'), m, f', v), \text{body}(\text{rep}(a'), m)) \text{cn}_\text{graph}\]
\[= cn''\]

where \(a_1(x) = \text{rep}(a)(x)\) and \(a_2(x) = \text{rep}(a')(x)\) for all \(x\) in \(\text{dom}(a), \text{dom}(a')\), respectively, and where \(a_1(f) = a_2(f) = (v, \varepsilon)\) if \(a''(f) = (v, o)\) for some \(a''\) in \(cn'\) and \(o\), and \(a_1(f) = a_2(f) = (\bot, \varepsilon)\) \(a''(f)\) has the shape \((\bot, o)\) for all \(a''\) in \(cn'\) and moreover some such \(a''\) exists. In this case \(a_1 = \text{rep}'(a)\) and \(a_2 = \text{rep}'(a')\) by prop. 10.1. Then
\[cn'' = \text{down}(cn, \text{rep})(o(\text{rep}'(o), \text{rep}'(a), u_0, \varepsilon, \varepsilon) \circ \text{rep}'(a'), u_0, \varepsilon, \varepsilon)\]
\[t(\text{rep}'(o), \text{rep}'(l)[f'/x], s)\]
\[t(\text{rep}'(o'), \text{locals}(\text{rep}'(a'), m, \text{rep}'(f), v), \text{body}(\text{rep}(a'), m)) \text{cn}_\text{graph}\]
\[= \text{down}(cn \circ(o, a) \circ(o', a') t(o, l[f/x], s)\]
\[t(o', \text{locals}(o', m, f, e_2(a, 1)), \text{body}(o', m)) f(f, \bot)\]
\[= \text{down}(cn \circ(o, a) \circ(o', a') t(o, l[f/x], s)\]
\[t(o', \text{locals}(o', m, f, e_2(a, 1)), \text{body}(o', m)) f(f, \bot)\]

as desired.
call-ext: Consider the following type 1 transition:
\[\text{bind } z. cn \circ(o, a) t(o, l, x = e_1!m(e_2); s)\]
\[\rightarrow \text{bind } z. cn \circ(o, a) t(o, l, s) c(o, o', m, v)\] (44)

where \(o' = e_1(a, l) \in \text{Ext} \) and \(v = e_2(a, 1)\). We calculate:
\[\text{down}(cn \circ(o, a) t(o, l, x = e_1!m(e_2); s))\]
\[= \text{down}(cn, \text{rep})(\text{down}(o(o, a), \text{rep})(\text{down}(t(o, l, x = e_1!m(e_2); s), \text{rep})\]
\[(\text{cn}_\text{graph}))\]
\[= \text{down}(cn, \text{rep}) \circ(o(\text{rep}(o), \text{rep}(a), u_0, \varepsilon, \varepsilon)\]
\[t(\text{rep}(o), \text{rep}(l), x = e_1!m(e_2); s) \text{cn}_\text{graph})\]
\[\rightarrow \text{down}(cn, \text{rep}) \circ(o(\text{rep}(o), \text{rep}(a'), f', m, \text{rep}(v), \text{init}(f', \text{rep}(a)), u_0, \varepsilon,\]
\[e_1(\text{call}(\text{rep}(o), \text{rep}(a'), f', m, \text{rep}(v), \varepsilon)) t(\text{rep}(o), \text{rep}(l)[f'/x], s)\]
\[(\text{cn}_\text{graph})\]
\[= \text{down}(cn, \text{rep})(\text{send}(\text{call}(\text{rep}(o), \text{rep}(a'), f', m, \text{rep}(v)), \varepsilon),\]
\[\circ(o(\text{rep}(o), \text{rep}(a'), f', m, \text{rep}(v)), \varepsilon)) t(\text{rep}(o), \text{rep}(l)[f'/x], s)\]
\[(\text{cn}_\text{graph})\]
\[= \text{down}(cn, \text{rep})(\text{send}(\text{call}(\text{rep}(o), \text{rep}(a'), f', m, \text{rep}(v)), \varepsilon),\]
\[\circ(o(\text{rep}(o), \text{rep}(a), u_0, \varepsilon, \varepsilon) t(\text{rep}(o), \text{rep}(l)[f'/x], s)\]
\[(\text{cn}_\text{graph})\]
\[= \text{down}(cn \circ(o, a) t(o, l, s) c(o, o', m, v)) .\]
Consider next the type 1 transition below:

\[
\text{bind } z.cn \text{ o(o,a) t(o,l, return e; s) f(f, ⊥) → bind } z.cn \text{ o(o,a) f(f,v) (45)}
\]

where \( l(\text{ret}) = f \) and \( v = \hat{e}(a,l) \). Again we calculate:

\[
down(cn \text{ o(o,a) t(o,l, return e; s) f(f, ⊥)}) = down(cn, rep)(o(rep(o), rep(a), u_0, ε, ε) t(rep(o), rep(l), return e; s) cn_graph) \\
\rightarrow down(cn, rep')(o(rep'(o), rep'(a), u_0, ε, ε) t(rep'(o), rep'(l))/l(\text{ret})), u_0, ε, ε) cn_graph) \\
= down(cn, rep')(o(rep'(o), rep'(a), u_0, ε, ε) cn_graph) \\
= down(cn \text{ o(o,a) f(f,v)})
\]

where in the second but last step we use the fact that

\[
rep' = rep[\hat{e}(rep(o), rep(l))/l(\text{ret})].
\]

get: Now consider the following type 1 transition:

\[
\text{bind } z.cn \text{ o(o,a) f(f, v) t(o,l, x = e.get; s) } \\
\rightarrow \text{ bind } z.cn \text{ o(o,a) f(f, v) t(o,l[v/x], s) (46)}
\]

where \( f = \hat{e}(a,l) \). Again we calculate:

\[
cn \text{ o(o,a) f(f, v) t(o,l, x = e.get; s) } \\
= down(cn, rep)(o(rep(o), rep(a), u_0, ε, ε) t(rep(o), rep(l), x = e.get; s) cn_graph) \\
\rightarrow down(cn, rep)(o(rep'(o), rep'(a), u_0, ε, ε) t(rep'(o), rep'(l)[v/x]/s) cn_graph) \\
= down(cn \text{ o(o,a) f(f, v) t(o,l[v/x], s)})
\]

new: The final case is for object creation:

\[
\text{bind } z.cn \text{ o(o,a) t(o,l, x = new C(e); s) } \\
\rightarrow \text{ bind } z, o'.cn \text{ o(o,a) t(o,l[o'/x], s) o(o', init(C, \hat{e}(a, l))) (47)}
\]

We obtain:

\[
down(cn \text{ o(o,a) t(o,l, x = new C(e); s) }) \\
= down(cn, rep)(o(rep(o), rep(a), u_0, ε, ε) \\
\text{ t(rep(o), rep(l), x = new C(e); s) cn_graph) } \\
\rightarrow down(cn, rep')(o(rep'(o), rep'(a), u_0, ε, ε) \\
\text{ t(rep'(o), rep'(l)[o'/x], s) o(rep'(o'), init(C, \hat{e}(rep'(a), rep'(l))))}, u_0, ε, ε) cn_graph) \\
= down(cn, rep')(o(rep'(o'), rep'(a), u_0, ε, ε) \text{ t(rep'(o), rep'(l[o'/x]), s) } \\
\text{ o(rep'(o'), rep'(init(C, \hat{e}(a, l))))}, u_0, ε, ε) cn_graph) \\
= down(cn \text{ o(o,a) t(o,l[o'/x], s) o(o', init(C, \hat{e}(a, l)))})
\]
where we switch from rep to rep′ and use (6) as usual. This completes the proof of statement 1.

2. We proceed now by cases on the type 2 transition. Suppose \( \text{down}(cn) \rightarrow cn'' \), and we find \( z', cn' \) to complete the diagram as stated in the lemma. As above we apply the rules to configuration subsets, to elide uses of ctxt-1. Rules in the type 2 instance of fig 5 are straightforward and left to the reader. For the rules in fig. 9 excepting call-send, ret-2, get-2, and new-2 we can choose \( z' = z \) and \( cn' = cn \), since by def. 9.12 and corollary 9.17, \( \text{down}(cn) \equiv cn'' \). For the four remaining cases (five, since call-send splits in two dependent on whether the called OID is internal or not), each case is obtained by reversing the arguments, i.e. proving that if the type 2 transition holds, depending on the rule application and shape of configurations, then also the type 2 transition holds. This completes the argument. \( \square \)
Appendix D

Paper 3: ABS-NET: Fully Decentralized Runtime Adaptation for Distributed Objects
ABS-NET: Fully Decentralized Runtime Adaptation for Distributed Objects

February 28, 2013

Abstract

We present a formalized, fully decentralized runtime semantics for a core subset of ABS, a language and framework for modelling distributed object-oriented systems. The semantics incorporates an abstract graph representation of a network infrastructure, with network endpoints represented as graph nodes, and links as arcs with buffers, corresponding to OSI layer 2 interconnects. The key problem we wish to address is how to allocate computational tasks to nodes so that certain performance objectives are met. To this end, we use the semantics as a foundation for performing network-adaptive task execution via object migration between nodes. Adaptability is analyzed in terms of three Quality of Service objectives: node load, arc load and message latency. We have implemented the key parts of our semantics in a simulator and evaluated how well objectives are achieved for some application-relevant choices of network topology, migration procedure and ABS program. The evaluation suggests that is feasible in a decentralized setting to continually meet both the objective of a node-balanced task allocation and make headway towards minimizing communication, and thus arc load and message latency.

1 Introduction

An important problem, made more relevant by recent interest in cloud computing, is how to decouple computational processes from the underlying physical infrastructure on which they execute. One motivation for decoupling processes in a distributed infrastructure is that it becomes possible to handle resource allocation at layers lower than the application layer. Potentially, tasks can then be performed at the physical machine most suited at the moment, continually meeting global system requirements for e.g. even utilization and task-local requirements such as a bounded response time.

An important problem, made more relevant by recent interest in cloud computing, is how to decouple computational processes from the underlying physical infrastructure on which they execute. One motivation for
such decoupling is to free applications from handling resource allocation issues, which can instead be handled in a transparent fashion using generic, application-independent mechanisms. Potentially, tasks can then be performed at the physical machine most suited at the moment, continually meeting global system requirements for, e.g., utilization, power consumption, or task-local requirements such as a response time.

We consider the problem of runtime adaptation of tasks in the context of a core subset of ABS [14], a language and framework for modelling distributed object-oriented systems developed in the EU FP7 HATS project.

Following our preceding work [8, 9], we construct a networked structural operational semantics for the ABS subset in rewriting logic style [6], where objects execute on network nodes connected point-to-point using asynchronous message passing links. We showed previously how object migration could be supported in an efficient, transparent, and robust (lock-free) manner using location independent routing. In the present work, we examine how adaptation can be performed in the networked model by a controller process running on each node.

To enable precise reasoning and experiments on adaptability, we define three central Quality of Services (QoS) objectives which a solution for runtime adaptation in our context can be assessed against: node load, arc load and message latency. We abstract from many practical, implementation-level concerns when interpreting these objectives in our setting. The load for a specific node at a specific time is simply the number of active tasks running on it. The load for a specific arc is the number of messages traversing the arc. The latency for a specific message is the number of hops needed to reach its destination. We then restrict our consideration of adaptability to the problem of how and when to migrate objects to achieve the objectives as well as possible, given a specific network topology, ABS program, and node-local procedure for managing migrations.

Using a simulator which implements the key parts of our semantics, we have investigated how well objectives are fulfilled for some application-relevant choices of network topologies, programs and migration procedures.

One potential application of our work is as a basis for a decentralized middleware system with very few dependencies and assumptions running on a networked ("cloud") infrastructure, that allows the provider to get high resource utilization when running resource-oblivious programs from a third party.

1.1 Contributions

We show that extending a general and reasonably practical object-oriented language (ABS) to execute in a network is feasible, and highlight the issues involved, from extending the semantics to concerns at implementation. The techniques apply to similar languages. Given that a language has a formal
semantics, the extension process can be carried out formally. We also investigate parts of the design space for distributed adaptability algorithms in our decentralized setting.

2 ABS Background

ABS [13] is a language and framework for modelling distributed object-oriented systems, developed in the FP7 HATS project. In contrast to design-oriented languages such as UML, ABS offers constructs for expressing concurrency and the possibility to execute models according to an operational semantics descended from Creol [15]. In contrast to foundational concurrency-oriented languages such as the $\pi$-calculus [17], ABS provides higher-level primitives that can be used to directly model object-oriented systems.

Core ABS [14] is a language which contains the main features of ABS: a functional level for expressing data structures and side-effect free internal computations of distributed objects, and an object level for expressing concurrent objects, and communication among such objects via method invocation. The object level defines syntax for interfaces, classes, methods, object creation and method calls, where there is inheritance among interfaces but not among classes. The object level is accompanied by a type system and a structural operational semantics which preserves well-typing. Hence, method invocations in Core ABS cannot go wrong at runtime for type-checked programs; when an object makes a call to a method $m$ using an object identifier $o$, there always exists an object associated with $o$, which is an instance of a class which implements an interface where $m$ is defined.

The runtime unit of concurrency in Core ABS is a concurrent object group (cog). A cog contains one or more runtime objects, which perform cooperative scheduling of tasks. We use a variant of Core ABS where a single object is the unit of concurrency rather than a cog, similar to the variant of Albert et al. [1]. The choice is motivated by our focus on network adaptability of individual objects and computation tasks, which becomes more complicated when objects in a group must perform intermittent synchronization. In this language variant, all individual objects can be interpreted as actors, having local store and communicating with the environment only via asynchronous message passing. Additionally, our language variant fixes a number of minor inconsistencies in the syntax and semantics of the original Core ABS, for example by prohibiting multiple return statements which could cause unexpected nonterminating behaviour. The language is described in detail at the accompanying website (http://www.csc.kth.se/~palmskog/abs-net/).

A fragment of a Core ABS program is given as an example in Figure 1. The CastNode interface defines a method aggregate, which, when called on
some object, is intended to perform a convergecast operation in the binary
tree rooted at that object. Specifically, this means that if an object imple-
menting CastNode is a leaf in the tree (an instance of class LeafNode), it
simply returns a locally known integer, but if the object has child nodes in
the tree (an instance of class BranchNode), aggregate is called on both of
those objects and the results are added to the local integer and returned. In
this way, the aggregate method for the object o always returns the aggre-
gate of all local values in the binary tree of objects rooted at o.

The implementation of the aggregate method in the program highlights
the use in Core ABS of futures as placeholders for results from asynchronous
method calls. The variables fLeft and fRight hold futures which ultimately
resolve to integer values, as indicated by their type declarations. In the right
hand side of the declarations of the futures, the delimiter ‘!’ between the
object variable name and the method name signifies asynchronous invoca-
tion, which always immediately returns a future. The usual dot delimiter
‘.’ signifies a synchronous invocation which blocks the caller until the final
result is returned without any intermediary.

Before returning the aggregate of the current object, the aggregate of
each child node is retrieved by appending .get to the variable holding the
respective future. Evaluations of assignments with this construct can be
blocking, unless an await statement was executed first with the future vari-
able involved, e.g. await fLeft; . Executing await when the associated
future has not yet been resolved does not force the caller into busy waiting;
if there are method invocations for the object waiting be processed, con-
trol can be changed to the corresponding process at the discretion of the
scheduler, and pass back to the original invocation later.

Informally, a Core ABS runtime configuration is a bag of objects and
futures equipped with unique identifiers, along with unprocessed method
invocations. An object in the bag has values for all variables defined in
its class, a queue of processes representing received method invocations,
and possibly an active process. Futures either have the value to which
they resolve, or a placeholder to indicate that no resolution is available.
When an asynchronous method invocation statement is executed, a method
invocation is added to the bag, ready to be consumed by the callee. In
contrast with actor languages such as Erlang and Rebeca [19], which provide
the traditional guarantee that messages from one actor to another are always
processed in the order they are sent, the Core ABS semantics does not
prescribe any particular order for processing method invocations. In effect,
the runtime environment provides an unbounded number of one-place buffers
that objects can use to communicate with objects for which identifiers are
known.

While interface names are proper type names in Core ABS, class names
are not, and are thus only used in object creation with the new keyword. For
example, the assignment CastNode nd = new LeafCastNode(0); creates a
interface CastNode {
    Int aggregate();
}

class LeafCastNode(Int val) implements CastNode {
    Int aggregate() {
        return val;
    }
}

class BranchCastNode(Int val, CastNode left, CastNode right) implements CastNode {
    Int aggregate() {
        Fut<Int> fLeft = left.aggregate();
        Fut<Int> fRight = right.aggregate();

        Int aggregateLeft = fLeft.get;
        Int aggregateRight = fRight.get;

        return val + aggregateLeft + aggregateRight;
    }
}

Figure 1: Core ABS example interface and classes.

LeafCastNode object with the val variable set to 0.

3 Network Model and Semantics

To reason about object adaptability to environmental conditions, we bring selected parts of the infrastructure of a distributed system into our model, namely, network endpoints and links. Endpoints and links are modelled as graph nodes and arcs with FIFO-ordered message queues, respectively. Conceptually, we consider a node to consist of an object layer, where local objects reside, and a node controller, which acts as a mediator between the environment and node-local objects, as illustrated in Figure 2. This node controller is not treated explicitly in the immediately related work [8, 9]. The dashed arrow in the figure signifies that an object identifier is known by another object and thus can be used for method invocation. The node controller also contains logic for decision-making on adaptability. Seen abstractly, adaptability here becomes the problem of deciding when and where to migrate objects to achieve QoS objectives—with the added constraint that all reallocations must be decided locally at each node.

In order to support location transparency the basic problem is to route messages correctly between objects that have no prior, mutual knowledge of where they are located. Many solutions have been examined in the literature,
including centralized or decentralized location servers, pointer chaining, and broadcast or multicast search. Sewell et al. [18] has an extensive discussion of the various approaches in the literature, and their relative merits.

We have previously proposed a novel approach to location transparency based on location independent (aka name independent) routing [8, 9], where the idea is to defer the maintenance of message routes to an explicit routing process executing independently of application level messaging. Adapted to the approach suggested here, a node controller executing on each network node is responsible for maintaining routing information by exchanging routing tables with neighbouring nodes in the network. This allows object migration to be supported in a transparent fashion with only modest extension to the runtime state.

3.1 Operational Semantics

We have defined a networked semantics for Core ABS based on our previous work [8, 9]. Adaptability features such as routing table exchange and object migration are modeled as nondeterministic events, with the node controller consisting of nothing more than a globally unique identifier and a routing table. We refer to the combination of the Core ABS functional layer, Core ABS object syntax, and the associated structural operational semantics described below as ABS-NET. We intend for the semantics to both guide implementation, by defining a baseline for retaining program runtime behavioural similar to Core ABS in a distributed setting, and provide opportunities for further theoretical analysis of specific adaptability strategies by refinement.
3.1.1 ABS-NET Runtime Configurations

In our semantics, an ABS-NET runtime configuration consists of two disjoint subconfigurations: one bag `net` representing the network with nodes and arcs, and one bag `cn` of all objects located at nodes in the network. The fact that a particular object is located on a particular node is not explicitly represented, but reflected behaviourally in the reduction rules.

The bag `net` consists of nodes `nd (u, τ)`, where `u` is an identifier assumed to be globally unique, and `τ` is a routing table, and arcs `ar (u, Q, u')`, where `Q` is an unbounded queue which buffers messages from node `u` to node `u'`.

The bag `cn` consists of objects `ob (o, a, p, q, Q_{in}, Q_{out}, Σ)`, where `o` is an identifier assumed to be globally unique, `a` is a store for values of instance variables, `p` is the active process (if any), and `q` is a bag of inactive processes. `Q_{in}` is the message input queue (mailbox) and `Q_{out}` is the message output queue; both queues are unbounded. We do not represent futures directly; instead, each object is equipped with a structure `Σ` that contains a map from future identifiers to values. When a future value is needed by an object, it must be put into the map, which happens after a certain message is delivered by the node controller to the object. The motivation for an implicit future representation is mainly to keep the semantics uniform and of manageable size; first-class futures would require extending routing and mobility beyond objects. In an implementation, globally unique object identifiers can be achieved during object creation by adding a serial number to the identifier of the node the new object will be located at.

Network and object behaviour is defined in rewriting logic, in the same style as for Core ABS. All reduction rules are implicitly specified modulo commutativity and associativity of configuration composition, with the empty configuration `ϵ` as unit. Rules either define how a whole configuration changes, or how a matching subconfiguration changes while the remaining part stays unchanged. The former case is distinguished by the use of brackets around the configuration in the rule, e.g. `{cn}`. The following is an example of a pair of configurations describing a two-node network with a single object located on one of the nodes:

\[
\{ \text{nd} (u, τ) \text{ ar} (u, Q, u) \text{ ar} (u, Q', u') \text{ nd} (u', τ') \text{ ar} (u', Q'', u') \} \text{ } \{ \text{ob} (o, a, p, q, Q_{in}, Q_{out}, Σ) \}
\]

The key rules for node controller and object behaviour are described below in Section 3.2 and Section 3.3, respectively. The complete set of language definitions and rules for both the Core ABS variant and ABS-NET are available on the accompanying website (http://www.csc.kth.se/~palmskog/abs-net/).
3.1.2 Handling Future Resolution

When representing futures implicitly as mappings stored in objects, there are at least two distinct strategies for how to ensure that future values are delivered through the network to the objects that need them, echoing the eager and lazy evaluation strategies in functional programming. In both strategies, the object which receives a message with a method call has an obligation to send back a message with the future value to the caller. The main difference between the strategies concerns what to do when an object shares a future identifier with another object by sending it as a parameter in a method call, as in the following Core ABS fragment:

```abs
Fut<Int> fArg = obj1!getArgument();
Fut<Int> fResult = obj2!getResult(fArg);
Int result = fResult.get;
```

The eager strategy assumes that all objects which receive a future identifier will attempt to retrieve the associated value; hence, whenever futures are transmitted, some action is taken which ultimately results in the resolved values of those futures being sent to the callee. In a lazy strategy, no particular future-related action is taken by the caller or any other entity. Instead, an object that actually needs a particular future value at run time requests it by sending special message, which is routed to the original object which originated the future identifier. The originating object then responds with a message containing the future value, once it becomes available.

Two drawbacks of an eager strategy are (1) that method call parameters must be inspected for occurrences of future identifiers, even when they are deeply nested in data structures, and (2) that messages containing future values are sent unnecessarily when callees simply ignore future identifier arguments. The lazy strategy avoids both of these drawbacks, but has drawbacks of its own. For example, retrieval of a future value, which can be costly and time-consuming, cannot happen concurrently with the processing that occurs before the future is needed. This means that if there is a sequence of blocking `get` operations without `await`, all actions to retrieve the values will happen serially.

We have chosen to use an eager strategy for future resolution in our semantics. Specifically, each object maintains a list of futures for which it is obligated to forward resolved future values, and where those resolutions should be sent. Whenever a future value is shared with another object, e.g. through the arguments of a method invocation, the list is updated accordingly. When the object retrieves a resolved future value, the value is saved and forwarding by message passing according to the list becomes possible. Because futures can resolve to other futures, the forwarding list may need to be updated when a new future value is added. One reason for selecting this approach is that it is reasonably straightforward to pinpoint where (i.e., in which reduction steps) the future list, or map, needs to be
updated, and what data needs to be considered.

3.1.3 Adaptability Assumptions

In the semantics, the Core ABS program being executed is assumed to be available unaltered at all nodes. The program is therefore not explicitly represented in the runtime configuration. Initially, we consider only networks that remain static over the course of program execution. We discuss briefly the case of benign dynamic networks, which is a planned extension, but leave all details of crash failures and byzantine failures for future work. On the same note, we also assume that messages sent between adjacent nodes cannot be lost—only ignored indefinitely as far as fairness permits.

We also assume that the behaviour of the program running on network nodes is nonterminating and cyclical. This assumption is motivated by our focus on adaptability; for adaptations to current conditions to have a chance of conveying benefits, similar conditions must hold in the future. Equivalently, if future conditions are random independently of current conditions, there is no obvious payoff in an adaptation strategy.

3.2 Node Controller Behaviour

The node controller’s relationship with the interpreter layer residing on the node is symbiotic. On one hand, the node controller provides message delivery services and callback functions to obtain new globally unique object identifiers for objects residing in the interpreter layer. On the other hand, the node controller triggers object movement by using callback functions that the interpreter layer makes available. We assume the node controller is aware, through its interaction with underlying network layers, of all nodes adjacent to the node it resides on, and can communicate with node controllers at neighbouring nodes. As mentioned earlier, in the model, such communication takes place through a buffer at an arc. In addition, each node controller is equipped with a self-loop arc that serves as the default route for messages that cannot immediately be routed to a neighbour node. Since there is no upper bound enforced on communication delays, the node controller always runs the risk that information received from the outside world is out of date.

Node controller behaviour is given in the semantics by the possible reductions on net configurations. There are two main kinds of reductions: labelled and unlabelled. Labelled reductions define how the node controller exchanges information with the interpreter layer, while unlabelled reductions define actions that can be taken independently of the interpreter state, such as sending and receiving routing messages. Similarly, interpreter behaviour is given by reductions on cn configurations, which are also labelled or unlabelled. The whole state of the system at any time is defined as a pair of
a net and a cn configurations, with a system evolution step being either an independent reduction by one member of the pair or a mutual transition on matching labels. The node controller transition rules related to transmission of routing tables (TABLE messages) are straightforward and unlabelled, with the condition $u' \neq u$ ensuring that node controllers do not send themselves messages through the self loop queue:

$$
\begin{align*}
(NET\_TABLE\_SEND) & \quad u' \neq u \\
Q & \quad \text{enqueue (TABLE ($\tau'$))} \\
\frac{\text{nd} (u, \tau) \text{ ar} (u, Q', u')}{\text{nd} (u, \tau) \text{ ar} (u, Q', u)} \\
\to & \quad \text{nd} (u, \tau) \text{ ar} (u, Q', u) \\
\end{align*}
$$

$$
\begin{align*}
(NET\_TABLE\_RECV) & \quad \text{update ($\tau', u'$)} \\
\frac{\tau}{\text{update ($\tau', u'$)}} \\
\frac{\text{ar} (u', Q, u) \text{ nd} (u, \tau) \text{ ar} (u', Q', u) \text{ nd} (u, \tau)}{(NET\_TABLE\_RECV) \quad \text{recv}} \\
\to & \quad \text{ar} (u', Q', u) \text{ nd} (u, \tau) \\
\end{align*}
$$

In contrast, a message $msg$ between objects (either a CALL message for a method invocations or a FUTURE messages for a resolved future) requires two labelled rules for sending and receiving it from the object layer, and one unlabelled rule, for routing between nodes when necessary:

$$
\begin{align*}
(NET\_MSG\_RECV\_OUT) & \quad \text{dest (msg) = o o} \\
\frac{\text{nd} (u, \tau) \text{ ar} (u', Q, u) \text{ dest (msg) = o o} \in \tau}{\text{nd} (u, \tau) \text{ ar} (u', Q, u) \text{ dest (msg) = o o} \in \tau} \\
\to & \quad \text{nd} (u, \tau) \text{ ar} (u', Q, u) \text{ dest (msg) = o o} \in \tau \\
\end{align*}
$$

$$
\begin{align*}
(NET\_MSG\_SEND\_IN) & \quad \text{dest (msg) = o o} \\
& \quad \text{next ($\tau$, o', u) = u'} \\
\frac{\text{nd} (u, \tau) \text{ ar} (u, Q, u')}{\text{nd} (u, \tau) \text{ ar} (u, Q, u')} \\
\to & \quad \text{nd} (u, \tau) \text{ ar} (u, Q, u') \\
\end{align*}
$$

$$
\begin{align*}
(NET\_ROUTE\_FURTHER) & \quad \text{dest (msg) = o o} \\
& \quad \text{next ($\tau$, o, u) = u''} \\
\frac{\text{nd} (u, \tau) \text{ ar} (u', Q_1, u) \text{ nd} (u, \tau) \text{ ar} (u', Q_1, u) \text{ nd} (u, \tau) \text{ ar} (u', Q_2, u'')}{\text{nd} (u, \tau) \text{ ar} (u', Q_1, u) \text{ nd} (u, \tau) \text{ ar} (u', Q_2, u'')} \\
\to & \quad \text{nd} (u, \tau) \text{ ar} (u', Q_1, u) \text{ nd} (u, \tau) \text{ ar} (u', Q_2, u'') \\
\end{align*}
$$

The condition $o \in \tau$ is to be interpreted as saying that the object $o$ is registered in the routing table $\tau$ as being located on the current node. Note also that, when an object sends a message to itself or some other object located on the same node, it will be the case that $u' = u$, and hence the self-loop arc will be used in the reduction. The auxiliary function dest simply returns the first argument in a message, which for the related message types is always the identifier of the destination object.

The rules for moving objects between node controllers are similar to those for passing messages to objects, with the main difference being that no routing is involved for deciding the direction. The rules permit objects being passed around between nodes indefinitely without ever letting the related tasks finish, but implementations will typically want to exclude such executions. Note that both the sender and the receiver needs to update their respective routing tables via auxiliary functions to reflect the new object allocation:
The semantics abstracts from concerns how the routing table $\tau$ is concretely represented, specifying its properties only with the help of the auxiliary functions update, next, register and replace. To achieve the possibility of convergence, implementations of these functions must fulfill various post-conditions, e.g., that old routes are replaced with new ones in the result of update.

The one remaining reduction rule for node controllers concerns creation of objects, which uses the label $\text{rg}$ (for ‘registration’):

$$
\frac{\text{(NET\_NEW\_OBJECT\_IN)}}{\begin{array}{l}
\text{fresh} \ (o') \quad o \in \tau \\
\tau \quad \text{register} \ (o', u, 0) \\
\end{array}} \quad \frac{\text{nd} \ (u, \tau) \ \text{ar} \ (u, Q, u')}{\text{rd} \ (u, \tau')}
$$

The condition $\text{fresh} \ (o')$ is meant to ensure that the new object identifier is globally unique, which translates to there being no such identifier registered in the routing tables of any node in the network. The condition $o \in \tau$ is needed to ensure that the object $o$ which spawned the new object is actually located on the node in question.

For the sake of an example of how the rules work, assume the object $o$ is located on the node $u$, and an object with identifier $o'$ is located on the adjacent node $u'$. Suppose $msg$ is a CALL message being sent from $o$ to $o'$. Suppose the queue $Q$ in the arc between the two nodes is initially empty, and that the queue $Q'$ denotes the queue that only contains $msg$. The following reduction sequence then describes how the partial state involving these nodes and the arc between them evolves, when the message is passed from $o$ to $o'$:

$$
\text{nd} \ (u, \tau) \ \text{ar} \ (u, Q, u') \ \text{nd} \ (u', \tau') \quad \frac{\text{tr} \ (o, msg)}{\text{nd} \ (u, \tau) \ \text{ar} \ (u, Q', u') \ \text{nd} \ (u', \tau')}
$$

Another example starting from the same state involves the object $\text{object}$ with identifier $o$ migrating from $u$ to $u'$. Let $Q''$ be the queue that only contains the message $\text{Object} \ (\text{object})$. We then get the following sequence of reductions:
Here, $\tau''$ is the routing table $\tau$ updated with the fact that $o$ is now found in the direction of $u'$, while $\tau'''$ updates $\tau'$ with the fact that $o$ is located on the current node $u'$.

### 3.3 Object Behaviour

The reduction rules in the Core ABS semantics which involve only a single object and its internal state have been transferred essentially unchanged into unlabelled interpreter layer reduction rules. For example, the following Core ABS reduction rule:

$$
(\text{cond\_true}) \\
\llbracket b \rrbracket a \circ l = \text{True} \\
ob (o, a, \{ \text{if } b \{ \pi \} \text{ else } \{ \pi' \} \pi \}, q) \\
\rightarrow ob (o, a, \{ \{ \pi \pi \}, q)
$$

is simply translated into the following rules in ABS-NET:

$$
(\text{abs\_cond\_true}) \\
\llbracket b \rrbracket a \circ l = \text{True} \\
ob (o, a, \{ \text{if } b \{ \pi \} \text{ else } \{ \pi' \} \pi \}, q, Q_{in}, Q_{out}, \Sigma) \\
\rightarrow ob (o, a, \{ \{ \pi \pi \}, q, Q_{in}, Q_{out}, \Sigma)
$$

In contrast, transitions that involve multiple objects or futures are translated into either transitions involving message passing or labelled rules for exchanging data with the node controller. For object creation, both the node controller rule $\text{NET\_NEW\_OBJECT\_IN}$ given above and the following rule for the interpreter layer need to be involved:

$$
(\text{abs\_new\_object\_out}) \\
\text{init } (C) = \text{process} \\
\llbracket \pi \rrbracket a \circ l = \pi \\
\text{atts } (C, \pi, o') = a' \\
\text{forwards } (\text{futsof } (\pi), o, \{ o' \}, \Sigma) = \Sigma' \\
ob (o, a, \{ l | x = \text{new } C(\pi); \pi \}, q, Q_{in}, Q_{out}, \Sigma) \\
\text{\texttt{let}\_\pi \pi \pi \rightarrow} ob (o, a, \{ l | x = o'; \pi \}, q, Q_{in}, Q_{out}, \Sigma') \\
ob (o', a', \text{idle, process, } (), (), [])
$$

When such a mutual transition has taken place, the new object has been properly added to the interpreter layer, and its globally unique identifier registered on the node of the object that spawned it. The init and atts auxiliary functions are unchanged from Core ABS, and construct the initial task of the object as given in the corresponding class definition, and
initializes variables based on given arguments, respectively. The auxiliary function forwards produces an update to the forwarding obligations in \( \Sigma \) to include futures in the argument list.

Given that the semantics abstracts from details on marshalling and just passes object states directly in messages, the interpreter layer rules for object mobility, which interact with the rules \texttt{NET_OBJECT_SEND_IN} and \texttt{NET_OBJECT_RECV_OUT} above, are very straightforward:

\[
\frac{\text{(ABS_OBJECT_SEND_OUT)}}{Q_{\text{out}} \stackrel{\text{enqueue}(\text{msg})}{\rightarrow} Q'_{\text{out}}}
\quad \frac{\text{(ABS_OBJECT_RECV_IN)}}{Q_{\text{in}} \stackrel{\text{enqueue}(\text{msg})}{\rightarrow} Q'_{\text{in}}}
\]

Although the rules for actually generating object-addressed messages are relatively complex due to the forwarding of futures, the rules for passing messages back and forth with the node controller are uncomplicated since eligible messages have been put in the out queue of the object:

\[
\frac{\text{(ABS_MSG_SEND_OUT)}}{Q_{\text{out}} \stackrel{\text{enqueue}(\text{msg})}{\rightarrow} Q'_{\text{out}}}
\quad \frac{\text{(ABS_MSG_RECV_IN)}}{Q_{\text{in}} \stackrel{\text{enqueue}(\text{msg})}{\rightarrow} Q'_{\text{in}}}
\]

4 Adaptation

We consider three QoS objectives which runtime adaptation solutions can be assessed against: node load, arc load and message latency.

In our setting, the definition of node load is simple but coarse grained: the load on a node \( u \) is the number of objects located on \( u \) with active tasks. One advantage of this measure is that it is an intrinsic property of runtime configurations, rather than something extrinsic to our model such as processor load or the \texttt{loadavg} measure available in many Unix operating system variants. We need a model-intrinsic measure of load to enable reasoning at an abstract level about convergence to balanced allocations and that loads stay within a certain range. One disadvantage of the approach is that it fails to take into account the varying use of memory and processing power among tasks. However, in an implementation, a more fine-grained measure of load can be adopted, as long as it is linear in the number of active tasks.

We define the load of a particular arc as the number of messages traversing it. Hence, global minimization of arc load means that a minimal number of inter-node messages are sent overall, with respect to the current state of routing tables at nodes. Unless all routing tables are optimal (minimum stretch), however, there is no guarantee that the number of hops, i.e., latency, of a particular object-addressed message is minimal.
4.1 Node Load Balancing

Although we wish to simultaneously meet all our QoS objectives fully, we consider node load balancing our primary concern. Load balancing solutions are also relatively well-studied in the literature, making it easier to find a good starting point.

Azar et al. [3] consider the problem of achieving balanced allocations in the framework of stochastic processes, where it is viewed as stepwise allocations of balls into bins. They highlight the use of greedy schemes for quickly converging to a ball-to-bin assignment where the maximum number of balls in any bin is minimized. The main drawback of this approach in a distributed setting is the reliance on atomic, single assignments of a ball to a bin at each algorithm step. Even-Dar and Mansour [11] study load balancing in a distributed setting where allocations are not necessarily done one-at-a-time. They give a distributed algorithm for selfish rerouting that quickly converges to a Nash equilibrium, which corresponds to a balanced resource allocation. However, at each round, locally computing a new allocation requires knowing precisely all loads in the system, which is complicated and costly to find out in the current setting.

Berenbrink et al. [4] describe and analyze fully distributed algorithms which require only local knowledge of the total number of resources and the load of one other resource to perform a single task migration step. The algorithms, some of which have attractive expected time for convergence, can be straightforwardly translated to a synchronous, round-based distributed setting, and further, e.g., via synchronizers [2], to a fully asynchronous setting. One important assumption made in the algorithm analysis is that a task can migrate to any other resource in a single concurrent round. For this property to hold, the underlying network graph must be complete, which we do not generally assume.

A factor in the convergence time is whether neutral moves are allowed, i.e., whether a migration can happen even when, as far as can be told locally, the move does not result in a more balanced allocation but merely an equally good one. If the network graph is sparse, and the number of active tasks an order magnitude greater than the number of nodes, allocations where the difference in load between any two neighbours is one but the maximal load difference is in the order of the graph diameter are possible. Such allocations clearly cannot be improved upon without neutral moves.

The problem of oscillating behaviour during task balancing can be mitigated by the use of coin flips before finalizing decisions to migrate tasks, as in the algorithms of Berenbrink et al. Oscillation can be made worse by information becoming stale, which is a fact of life in asynchronous systems. If the information is not too stale, however, the number of oscillation periods can sometimes be bounded [12].
4.2 Minimizing Communication and Other Objectives

The literature on load balancing related to scientific computing contains work on simultaneously optimizing task allocations and communication overhead. For example, Cosenza et al. [7] give a distributed load balancing scheme for simulations involving agents moving in space from worker to worker. The scheme, which is validated experimentally, optimizes both worker load and communication overhead between workers, but assumes only a small area of interest for each agent, with agents unable to communicate with other agents outside this area. In the current work, objects can communicate whenever object identifiers are known to the sender, making it harder to minimize communication overhead. Catalyurek et al. [5] describe how to use hypergraph partitioning to minimize both communication volume and migration time of tasks for parallel scientific computations. However, the repartitioning is performed in batch and requires complete, immediate knowledge of the data and computations on each node.

Querying the load of neighbours before deciding where to migrate an object can be costly in terms of arc load, and information received previously may not be accurate. Many load balancing algorithms therefore have as a feature that the number of load queries sent is minimal when migrating a resource. A third measure which is discussed in the literature which we do not consider is the cost in terms of time and messaging for migration itself.

5 Evaluation

In addition to the theoretical results on ABS-NET described elsewhere [8, 9], we have evaluated ABS-NET by developing a simulator for running ABS programs in a network of nodes according to our semantics. We have run the simulator with a variety of network node topologies, programs and object migration policies.

5.1 Simulator

Our simulator’s main purposes are to serve as a proof-of-concept for ABS-NET, and to allow us to run various adaptability case studies with particular programs and topologies. Specifically, we are interested in studying convergence properties of object migration policies in practice, and in showing that our approach of distributed execution scales to networks with many nodes. There are several other ways of executing ABS programs developed in the HATS project [10], but the main feature we need that is absent from all of them is object mobility between nodes or sites. Also, in contrast with most of these ABS backends, which aim to provide an execution platform for the full ABS language, the simulator only supports a subset of the Core ABS language; notably, the await statement is not supported.
The simulator is implemented in Java. Each node controller is implemented as a Java thread, which communicates with other controllers through TCP sockets, using the KryoNet network library [16]. One reason for choosing to use sockets is to enable to scale simulations over several physical machines and a large number of simulated network nodes. All node controllers in the network have a representation of the abstract syntax tree of the ABS program being executed, which is generated from ABS program code by the lexing and parsing frontend shared by most ABS backends.

As in the conceptual model and the formal semantics, a node controller can have zero or more objects, each having at most one active task. An active task has a reference to the statement currently being executed in the abstract syntax tree. We call an object active if it has an active task. Scheduling of active tasks is done at the node controller level in a round-robin fashion for active objects. More precisely, the scheduler deterministically steps all active tasks, checks for active objects, and then repeats the process on the new set of active tasks.

We implement statement execution by interpretation. The main reason for this choice is to enable easy serialization of objects between executing statements; to get immediate results from load balancing, we must be able to migrate active objects. One drawback of using interpretation is that local execution is slow and resource-demanding compared to the standard ABS backends.

A node controller is associated with a unique TCP port on the host system. Besides a list of neighbour handles, which abstract over underlying sockets, and a list of local objects, the node controller maintains a routing table. The routing table is broadcast to neighbours after entries have been changed or added as a result of statement execution or incorporation of routes from neighbour messages. Hence, except after a short interval with many updated locations, we expect routing tables to be up-to-date or nearly so. The node controller also stores incoming messages that cannot be processed locally or rerouted.

Network topology setup and program loading is handled by scripting on top of a custom simple command-line interface (CLI). When starting up, a node controller is assigned a migration policy through the CLI, which is assumed to be the same for all node controllers in the network. A migration policy is based on one of the adaptation strategies described below.

By default, the simulator starts the initial task of the initial object on a single startup node. In all our programs, the initial task creates all the objects used for the duration of the program. Migration and logging does not commence until a method with the name \texttt{setupFinished} is called on some object. There are several reasons for this kind of initialization; it is easier to predict load balancing behaviour with a fixed set of objects, and it is problematic to create new objects on the fly without proper distributed garbage collection, which we have not implemented.
One desirable feature that the simulator currently lacks is control of link characteristics, such as delays.

5.2 Scenarios

There are many parameters to consider when setting up interesting scenarios for studying adaptation via simulations, as outlined below.

Network configuration The size and topology of the network. Large and dense networks obviously give more overhead in the form of messaging (e.g. routing and load), making simulations slower.

Object behaviour The number of objects generated by the program, inter-object communication patterns, and the fraction of objects with active tasks over time. In practice, this means selecting the appropriate ABS program and adjusting some method parameters.

Adaptation strategy This includes both the logic for deciding when and where to migrate objects, and for messaging to exchange information used as basis for decisions.

By necessity, we have explored only a small cross-section of the possible parameters, at this initial stage of the work.

5.2.1 Network Configurations

The possible sizes and configurations of networks to be simulated are limited by the performance of the prototype simulator. Currently, highly connected topologies with in the order of 25 network nodes can be simulated in reasonable time. On this note, we limit the evaluation to networks with three distinct underlying network topologies for nodes along the continuum from sparsely to fully connected: grids, hypergraphs and full meshes. Our base initial setup for each topology has 32 nodes.

Since the simulator scales to at least in the order of 100 nodes for sparsely connected topologies, we also investigate grids larger than 32 nodes for some scenarios to compare results.

5.2.2 Benchmark Programs

We have developed a number of ABS programs specifically to run in our simulator. All programs share the characteristic that they have a setup phase, where a fixed number of objects are initialized, and a phase where the generated objects perform some computation, possibly involving communication; there are no short-lived dynamically created objects. For all programs but one, which implements a distributed hash table (DHT) algorithm, communication patterns among generated objects follows straightforwardly from
the code. This makes it easier to follow what happens during a simulation and to reason about how far an allocation of objects to nodes is from the optimum. After running initial simulations, we have adjusted parameters in our programs, and in some cases added functionally redundant instructions, to get constant and reasonably consistent load and messaging, since our migration procedures consider mainly objects with active tasks. With spurious activity among nodes, messaging and load varies greatly, and progress over time becomes hard to discern. Sometimes this is due to behaviour inherent to the program, as in the convergecast program described in Section 2, which gives rise to periodic bursts of messages. We focus on programs with more consistent behaviour.

**IndependentTasks.abs** The starting task generates objects, and each generated object is called upon to perform a long-running task. There is no communication among workers—only briefly at startup between the coordinator object, which initializes and assigns tasks, and the generated objects. Since there is no communication, an optimal allocation is simply a completely even distribution of objects to nodes, regardless of the network topology.

**Ring.abs** The starting task generates objects which know the identifiers of the next object in the ring. The last object generated gets the identifier of the first object. The first object, when called, calls its next object, and so on, until the object which has the first object as next object is reached. In the computation phase, many such calls traverse the ring simultaneously.

**Star.abs** An object star configuration consists of a center object and one or more fringe objects. The fringe objects in the star continually communicate with the center object, but not among themselves. The program builds a number of independent of object star configurations.

**ChordDHT.abs** An implementation of the Chord DHT algorithm [20]. Key-value mappings are distributed between a number of nodes, which all support a put/get interface to clients. Nodes are arranged in a ring, but aside from references to their neighbours, each node has log(n) “fingers” to non-adjacent nodes, where n is the size of the keyspace. The addition, or join, of a node to the ring places the new node at a particular position based on its identifier and can trigger global re-configuration of the ring. During setup, 128 nodes are joined to the chord, and each node becomes associated with either a producer object, which continually puts values into the DHT, or a consumer object, which continually attempts to retrieve values from the DHT.
5.2.3 Adaptation Strategies

We have generally restricted ourselves to strategies that as a first priority balance out load evenly among nodes in the network. As a consequence, a simulated node controller continually informs neighbours nodes of its load when appropriate, and receives load messages from neighbours in turn, regardless of the migration procedure used.

In the simulator, each migration policy defines a callback method which takes the affected node controller as a parameter. The callback method is invoked, and can possibly result in the migration of several objects to neighbour nodes.

Berenbrink et al. An adapted version of the selfish distributed load balancing algorithm by Berenbrink et al., which does not allow neutral moves. One notable difference in the simulator implementation from the abstract description given in Algorithm 1 is that only a fixed small number of objects (20) have the possibility to migrate in each cycle, because of restrictions in the size of message buffers.

Berenbrink et al. with neutral moves An adapted version of the selfish distributed load balancing algorithm by Berenbrink et al., which does allow neutral moves, and therefore is only expected to converge to a completely stable state after a long time, exponential in the size of the network. As determined experimentally, only migrating one or two objects per cycle leads to significantly less oscillation of objects than when directly implementing the abstract description given in Algorithm 2.

Berenbrink et al. with communication intensity A variant of the preceding policy, where objects are selected for migration based on their affinity to the (randomly) chosen neighbour node, as determined by their communication history with objects in the neighbour node’s direction. The communication history is a list of other objects that a given object has communicated with recently, as given by abstract object-local time, defined by the number of tasks finished since initialization. The affinity of an object to the neighbour node is then quantified as the number of objects in the communication history that are located in the direction of the node, according to the routing table.

Weighted neighbour load difference Once every cycle, an object and an adjacent node are chosen uniformly at random and independently. Then, a probability of migration is calculated and enacted based on the difference in load between the current node and the chosen node, with probability 1 for a difference of 10 or more, and probability 0 for a negative difference. Specifically, if the load difference is $d$, the
probability of migration becomes $\frac{d}{10}$, adjusted to closest number in the interval $[0,1]$.

**Weighted neighbour load difference with communication** Given a randomly chosen object and adjacent node as in the previous policy, we define the probability of migration according to communication intensity as the number of entries in the object’s communication history found in the direction of the node, divided by the total number of entries in the history. This probability is then combined via weighted averaging with the neighbour load difference probability to define the weighted neighbour load with communication policy. We have used the weight $0.2$ for the communication intensity probability and $0.8$ for the neighbour’s load probability.

---

**Algorithm 1** Berenbrink et al. load balancing cycle.

```plaintext
for each active object o do
    u' is a neighbour chosen uniformly at random
    l is the current load
    l' is the last known load of u'
    if $l > l' + 1$ then
        send o to u' with probability $1 - l'/l$
    end if
end for
```

---

**Algorithm 2** Berenbrink et al. load balancing with neutral moves cycle.

```plaintext
for each active object o do
    u' is a neighbour chosen uniformly at random
    l is the current load
    l' is the last known load of u'
    if $l > l'$ then
        send o to u' with probability $1 - l'/l$
    end if
end for
```

---

### 5.3 Scenario Objectives

Since our primary objective is to balance node load evenly, we record the load of all individual nodes over time, and then show maximum load and load standard deviation. For scenarios with little to no object communication, these are the only measures that are relevant with respect to our objectives. For scenarios with significant messaging, we also consider the number of object-related messages sent (i.e., CALL and FUTURE messages) by each
node between sampling intervals—with the average number of messages and standard deviation shown. We do not count messages sent by a node to itself via the self-loop arc, since such messages need not go through a physical link in an implementation.

We sample the required quantities from simulations at a fixed global rate, corresponding roughly to a certain number of transitions (1000) in the semantics with imposed fairness via round-robin scheduling. The imposed fairness provides a degree of synchrony in the simulated network.

5.4 Results

Below, we give an overview of the results from our simulations of the scenarios described in the previous sections.

5.4.1 Simulations of IndependentTasks.abs

The program creates 201 objects in total: one starting object which becomes inactive after initialization and 200 objects that each have a task that runs for the course of the program.

As expected, the algorithm by Berenbrink et al. without neutral moves converges very quickly and stays unchanged with no migrations after reaching a state where neighbour load differences are at most one. For most of the runs on a 32-node hypergraph network topology, the stable state coincided with a completely balanced allocation, or very closely so. For the case of a 32-node grid, the stable allocation was in almost all cases some distance from a fully balanced one.

The algorithm variant with neutral moves and two migrations per cycle converges to an almost-stable state quite quickly on a hypergraph, but continues to have minor oscillation of objects. With the same algorithm but five migration allowed per cycle, there is considerably more oscillation going on after coming close to a balanced allocation. On a grid topology, where a stable allocation can be further away from a balanced allocation, allowing neutral moves gives better results than disallowing them, as expected.

The maximum load and the standard deviation of the load over time for the 32-node hypergraph network topology is shown in Figure 3 and Figure 4, respectively. The corresponding measures over time for an $8 \times 4$ grid topology are shown in Figure 5 and Figure 6, respectively. For a grid, the gain from using neutral moves is most distinctly recognized in the lower standard deviation compared to the algorithm without neutral moves in Figure 6. Results for running the program on a 32-node complete network graph are essentially the same as for the hypergraph case, and therefore omitted.
Figure 3: Global max load in hypergraph for \textit{IndependentTasks.abs}.

Figure 4: Load std. deviation in hypergraph for \textit{IndependentTasks.abs}.

5.4.2 Simulations of \textit{Star.abs}

In the star program, we construct stars precisely so that each node can hold a whole star, and there is precisely one node per network node. In an
Figure 5: Global max load in grid for IndependentTasks.abs.

optimal allocation, therefore, there are no node-to-node message exchanges at all; all messages are sent locally.
We expected the pure load balancing policies to have markedly worse results than the policies taking inter-object communication intensity into account. In Figure 7, the standard deviation of the number of sent messages over time is shown. In Figure 8, the average number of sent messages over time is shown. In both cases, the measurements have been smoothed out via averaging over five samples to reduce noise. As can be seen in the figures, there is a distinct improvement with respect to messages sent when using the algorithm by Berenbrink et al. augmented with message intensity comparisons when compared to the other policies, although it is quite far from the optimum. The algorithm using probabilistic weighting of load and messaging seems to improve the most over time, although it performs similarly to the messaging-augmented load balancing algorithm by Berenbrink et al.

With all the tested migration strategies for the hypergraph, load became evenly balanced relatively quickly, as seen in Figure 9, similarly to hypergraphs when running `IndependentTasks.abs`. Hence, there was no significant avoidance of messaging by communicating objects clustering at a few specific nodes.

Because of the simplicity of the object communication graph and the fact that it is possible to reach an allocation where no inter-node communication takes place, it is worthwhile to illustrate how near specific algorithms can get after many (1000) cycles, for comparison. In a given allocation, each object has a total distance in hops to the other object it communicates with. For fringe objects, the total distance is the number of hops to its center object, but center objects have total distance equal to the sum of all distances to its fringes. In an optimal allocation, all centers (and all fringes) have total distance zero. In Figure 10, gray bars show the distribution of total distance among the 32 center objects for the load balancing algorithm by Berenbrink et al. The black bars show the distribution of total distances of the objects for the algorithm by Berenbrink et al. augmented with message intensity comparisons. The distributions intersect, but the former algorithm fares distinctly worse, with most centers in the 10–20 distance range, while the latter algorithm has most centers in the 6–12 distance range.

### 5.4.3 Simulations of `Ring.abs`

When running a ring of 129 objects on a 32-node grid, there are balanced allocations where all but one node have 4 objects, where all objects that communicate are on either the same node or adjacent nodes. The idea is that two of the objects on a node are part of a segment of the ring, while the other two are part of another segment coming back the other way. Such allocations lead to few inter-node messages being needed for a method invocation that involves the whole ring. Almost all objects then have a combined distance of one to the objects they communicate with.
Figure 7: Std. deviation of sent messages for hypergraph in Star.abs.

Figure 8: Avg. sent messages for hypergraph in Star.abs.

In Figure 12, the standard deviation of messages of a 129-object ring for a grid topology is shown. Here, both the solutions which take message intensity into account show considerable improvement over time. This is also reflected in the average number of messages sent over time shown in Figure 13. The eventually lower number of messages sent are not due to clustering of many objects on a few nodes, as shown by the eventually low standard deviation for all migration strategies in Figure 11.

In Figure 14, gray bars show the distribution of total distance among all ring objects on a grid to the objects they communicate with, after 1000
migration cycles using the algorithm by Berenbrink et al. Black bars show the distribution for the algorithm by Berenbrink et al. augmented with message intensity comparisons. There is overlap, but the latter algorithm results in many more objects with total distance between 1 and 5. However, both distributions are quite far from being optimal.
5.4.4 Simulations of ChordDHT.abs

In the setup phase, a number of ServiceObject nodes are created and joined. Then, every such object becomes associated with either a producer object, which puts values into the DHT via the put method, or a consumer object, which tries to retrieve them via the lookup method.

Figure 15 shows the standard deviation of the number of messages sent for nodes when running the program on a hypergraph, and Figure 16 shows the average number of messages. In both figures, the curves have been smoothed out by averaging samples five at a time. The weighted neighbours load and message intensity strategy exhibited a tendency to quickly cause message buffer overflows for this program, which is why we do not show any results for it. The figures confirm that there is a reasonable payoff from
taking messaging into account in a migration strategy, even when running a program with relatively complex communication patterns.

5.4.5 Simulations of Star.abs on 64-Node Grid

We expected the improved performance of algorithms which consider message intensity to be greater on larger networks. We therefore investigated how the star program performed on a 64-node grid, with the number of star centers scaled up appropriately. The resulting program is called Star64.abs. The results were largely as expected, with a larger discrepancy between the pure load balancing procedure as compared to procedures which take mes-
saging into account, as shown in Figure 17 and Figure 18.

6 Conclusions and Future Work

The evaluation suggests that it is feasible in a decentralized setting to meet the objective of balanced resource allocation, and also make headway towards the objective of minimizing communication of distributed objects. The main concern for our results being valid for real-world networks is the use in our network model of unbounded message queues, and the lack of rate limitation and latency controls in our simulator.

In future work, we plan to continue the theoretical and simulation-based studies to deepen our understanding of multi-dimensional resource man-
agement, to improve the performance and accuracy of the simulator, and to investigate adaptation in dynamic networks, initially only with benign churn, i.e., with controlled startup and shutdown of nodes.
References


